12-2004

MultiJava: Design Rationale, Compiler Implementation, and Applications

Curtis Clifton
Iowa State University

Todd Millstein
Iowa State University

Gary T. Leavens
Iowa State University

Craig Chambers
Iowa State University

MultiJava: Design Rationale, Compiler Implementation, and Applications

Abstract
MultiJava is a conservative extension of the Java programming language that adds symmetric multiple dispatch and open classes. Among other benefits, multiple dispatch provides a solution to the binary method problem. Open classes provide a solution to the extensibility problem of object-oriented programming languages, allowing the modular addition of both new types and new operations to an existing type hierarchy. This paper illustrates and motivates the design of MultiJava and describes its modular static typechecking and modular compilation strategies. Although MultiJava extends Java, the key ideas of the language design are applicable to other object-oriented languages, such as C# and C++, and even, with some modifications, to functional languages such as ML. This paper also discusses the variety of application domains in which MultiJava has been successfully used by others, including pervasive computing, graphical user interfaces, and compilers. MultiJava allows users to express desired programming idioms in a way that is declarative and supports static typechecking, in contrast to the tedious and type-unsafe workarounds required in Java. MultiJava also provides opportunities for new kinds of extensibility that are not easily available in Java.

Keywords
Open Classes, Open Objects, Extensible Classes, Extensible External Methods, External Methods, Multimethods, Method Families, Generic Functions, % so people searching on this term can find it Object-oriented Programming Languages, Single Dispatch, Multiple Dispatch, Encapsulation, Modularity, Static Typechecking, Subtyping, Inheritance, Java Language, MultiJava Language, Separate Compilation, Expression Problem, Binary Method Problem, Augmenting Method Problem

Disciplines
Programming Languages and Compilers

MultiJava: Design Rationale, Compiler Implementation, and Applications

Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers

TR #04-01b
December 2004

Revised version of TR #04-01, dated January 2004 and titled
MultiJava: Design Rationale, Compiler Implementation, and User Experience

Keywords: Open Classes, Open Objects, Extensible Classes, Extensible External Methods, External Methods, Multimethods, Generic Functions, Object-oriented Programming Languages, Single Dispatch, Multiple Dispatch, Encapsulation, Modularity, Static Typechecking, Subtyping, Inheritance, Java Language, MultiJava Language, Separate Compilation

2002 CR Categories: D.3.1 [*Programming Techniques*] Object-oriented Programming; D.3.2 [*Programming Languages*] Language Classifications — object-oriented languages; D.3.3 [*Programming Languages*] Language Constructs and Features — abstract data types, classes and objects, control structures, inheritance, modules, packages, patterns, procedures, functions and subroutines; D.3.4 [*Programming Languages*] Processors — compilers; D.3.m [*Programming Languages*] Miscellaneous — generic functions, multimethods, open classes.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

MultiJava: Design Rationale, Compiler Implementation, and Applications

CURTIS CLIFTON
Iowa State University
TODD MILLSTEIN
University of California, Los Angeles
GARY T. LEAVENS
Iowa State University
and
CRAIG CHAMBERS
University of Washington

MultiJava is a conservative extension of the Java programming language that adds symmetric multiple dispatch and open classes. Among other benefits, multiple dispatch provides a solution to the binary method problem. Open classes provide a solution to the extensibility problem of object-oriented programming languages, allowing the modular addition of both new types and new operations to an existing type hierarchy. This paper illustrates and motivates the design of MultiJava and describes its modular static typechecking and modular compilation strategies. Although MultiJava extends Java, the key ideas of the language design are applicable to other object-oriented languages, such as C# and C++, and even, with some modifications, to functional languages such as ML.

This paper also discusses the variety of application domains in which MultiJava has been successfully used by others, including pervasive computing, graphical user interfaces, and compilers. MultiJava allows users to express desired programming idioms in a way that is declarative and supports static typechecking, in contrast to the tedious and type-unsafe workarounds required in Java. MultiJava also provides opportunities for new kinds of extensibility that are not easily available in Java.

Categories and Subject Descriptors: D.3.1 [**Programming Techniques**]: Object-oriented Programming; D.3.2 [**Programming Languages**]: Language Classifications—*object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types*; *classes and objects*; *control structures*; *inheritance*; *modules, packages*; *patterns*; *procedures, functions, and subroutines*; D.3.4 [**Programming Languages**]: Processors—*compilers*; D.3.m [**Programming Languages**]: Miscellaneous—*method families*; *multimethods*; *open classes*

General Terms: Languages, Design

Additional Key Words and Phrases: Open Classes, Open Objects, Extensible Classes, Extensible External Methods, External Methods, Multimethods, Method Families, Generic Functions, Object-oriented Programming Languages, Single Dispatch, Multiple Dispatch, Encapsulation, Modularity, Static Typechecking, Subtyping, Inheritance, Java Language, MultiJava Language, Separate Compilation, Expression Problem, Binary Method Problem, Augmenting Method Problem

1. INTRODUCTION

In this paper we describe the design and implementation of MultiJava [Clifton et al. 2000; Clifton 2001] and the ways in which the language has been used

by others [Millstein 2003]. MultiJava is a backward-compatible extension to the Java programming language [Gosling et al. 2000] that supports symmetric multiple dispatch and open classes. MultiJava is backward compatible with Java in two ways. First, existing Java programs are legal MultiJava programs and have the same meaning. Second, code using MultiJava's new language features interoperates with existing Java source and bytecode. Our MultiJava compiler is available from `www.multijava.org`.

1.1 The Problem

We begin by describing two problems that arise with mainstream object-oriented programming languages like C++ and Java [Stroustrup 1997; Arnold et al. 2000; Gosling et al. 2000]. These problems are examples of a general extensibilty problem that arises from not being able to dynamically dispatch on a class except by editing the class in place.

1.1.1 *The Binary Method Problem.* A well-known problem in mainstream object-oriented programming languages concerns expressing the behavior of methods when that behavior should vary with the dynamic types of more than one argument. Traditionally, the problem is considered in terms of *binary methods*, two-argument methods where the argument types should vary together. In the `Shape` class of Fig. 1, the method for calculating the intersection of two shapes—the receiver object `this` and the argument object `s`—is a binary method. Suppose that one wishes to create a class `Rectangle` as a subclass of `Shape`. When intersecting two rectangles, one can use a more efficient algorithm than for arbitrary shapes. We would like to implement this more efficient algorithm for rectangles, without having to modify existing code, in either the `Shape` class or its clients. Further, we would like to do this in a way that can be statically typechecked for safety.

Unfortunately there is no straightforward way to do this, because mainstream object-oriented languages cannot safely use subtypes in the argument positions of overriding methods. Doing so would violate the standard restriction against covariant subtyping for function parameter types [Cardelli 1988]. Thus, such languages cannot easily specify overriding binary methods for cases where both the receiver and non-receiver arguments are subtypes of the original types. This difficulty in specifying specialized behavior for overriding binary methods is known as the *binary method problem* [Bruce et al. 1995; Castagna 1995].[1]

In Section 2.1 we demonstrate several partial solutions to the binary method problem in Java and show how the problem can be solved with multiple dispatch in MultiJava.

1.1.2 *The Augmenting Method Problem.* Another well-known challenge in programming language design is to simultaneously support both the easy addition of new types and the easy addition of new operations to an existing type hierarchy [Reynolds 1975; Cook 1991; Odersky and Wadler 1997; Krishnamurthi et al.

[1]Another part of the binary method problem is that, in some languages, a receiver object of a binary method cannot easily take advantage of the private representation of the non-receiver argument. This part of the problem does not arise in Java, because in Java's encapsulation model all instances of a class have access to the private representation of all other instances of the class.

```
public class Shape {
  ...
  public Shape intersect(Shape s) {
    ...
  }
}
```

Fig. 1. A simple class with a binary method, `intersect`

1998; Findler and Flatt 1999; Garrigue 2000; Zenger and Odersky 2001]. For example, suppose we have a program with a collection of types representing various shapes—like `Shape`, `Rectangle`, and `Triangle`—and a collection of operations on these types, like `area` and `intersect`. We would like to easily add both new shape types and new shape operations to the program. Furthermore, both of these forms of extension should be expressible without having to modify the existing types or their existing clients.

In traditional functional and procedural languages, each operation is implemented separately from its associated type hierarchy. Thus, it is easy to add new operations to an existing type or type hierarchy. But there is no support for subclassing, so the addition of new types requires all existing operations on the associated type hierarchy to be updated in place. Conversely, in traditional object-oriented languages it is easy to modularly add new types to an existing type hierarchy via subclassing, and overriding methods allow existing operations on that hierarchy to be easily updated. However, each method must be declared inside its associated class, so there is no support for the modular addition of new operations; each existing class must be updated in place to contain a new method.

These problems of adding new types in functional or procedural languages and new operations in object-oriented languages are dual cases of what Wadler termed the *expression problem*.[2] We focus on object-oriented programming and its traditional data-centric encapsulation. We call the need for non-modular editing to add new operations in object-oriented languages the *augmenting method problem*.

In Section 2.2 we demonstrate attempts to solve the augmenting method problem in Java and a solution using MultiJava's open classes.

1.1.3 *Object-Oriented Extensibility.* The problems described above are examples of a more general problem with current mainstream object-oriented programming languages. In these languages it is impossible to dynamically dispatch on a class *externally*, that is without modifying the class in place. This restriction makes it awkward at best, and error-prone at worst, to extend an existing class in commonly desired ways. As we demonstrate, in current object-oriented languages such class extensions are often impossible without advance planning by the original programmer.

In MultiJava we lift this restriction of mainstream object-oriented programming languages, allowing new methods to dynamically dispatch on existing classes without modifying existing code. Multiple dispatch allows new methods to dispatch on existing classes at argument positions other than the receiver, and open classes

[2]Wadler coined this term in a 1998 discussion on the Java Generics mailing list.

allow new methods to dispatch on existing classes at the receiver position. MultiJava is the first full-scale programming language to provide these features while including modular, static typechecking and compilation.

The design of MultiJava satisfies the following goals and constraints:

— MultiJava provides complete backward compatibility with the extant Java language. Code written in Java has the same semantics when compiled with a Java compiler or a MultiJava compiler, including code that relies on Java's static overloading. It is possible to extend existing classes and override existing methods using MultiJava's new features.

— MultiJava retains Java's encapsulation properties.

— The modular static typechecking and compilation properties of Java are maintained.

— To allow for wide use of code written in MultiJava, output of the MultiJava compiler targets the standard Java Virtual Machine.

— For regular Java code the bytecode produced by the MultiJava compiler is no less efficient than that generated by a standard Java compiler. For source code using MultiJava's multiple dispatch or open classes, the bytecode produced by the MultiJava compiler has efficiency comparable to that produced by a standard Java compiler for Java code that simulates these features.

1.2 Outline

Section 2 introduces and motivates the design of MultiJava. The language is a small syntactic extension of Java, but it provides significant new opportunities for code organization and reuse. Section 3 describes MultiJava's static type system, which safely augments Java's modular (per-compilation-unit) typechecking. Section 4 sketches the translation of MultiJava source code into standard Java bytecode, again on a per-compilation-unit basis as in Java. MultiJava is being used by others in a variety of application domains; Section 5 describes the applications and programming idioms in which MultiJava's features have been successfully employed. Section 6 describes an extension to MultiJava that was sparked in part by user feedback. Section 7 presents some performance comparisons; Section 8 compares MultiJava to related work; and Section 9 concludes with a discussion of future work.

2. LANGUAGE DESIGN AND MOTIVATION

In this section we describe the two main language features—multiple dispatch and open classes—that allow MultiJava to solve the binary method and augmenting method problems, and the more general extensibility problem, introduced in Section 1.1. For each language feature we present a concrete example of the problem. We motivate our design by demonstrating how a Java-based approach is inadequate. Then we show how MultiJava provides a solution.

2.1 Multiple Dispatch

In Java [Arnold et al. 2000; Gosling et al. 2000], the method invoked by a call depends on the runtime type of the receiver argument, but it does not depend on the runtime types of any other arguments. This method selection scheme is known as

```
public class Rectangle extends Shape {
  ...
  public Shape intersect(Rectangle r) {
    /* efficient code for two Rectangles */
    ...
  }
}
```

Fig. 2. An attempt at implementing `Rectangle` using static overloading

single dispatch. Single dispatch is also found in Smalltalk, C++, and C# [Goldberg 1984; Stroustrup 1997; Troelsen 2003]. In constrast, *multiple dispatch*—found in Common Lisp, Dylan, and Cecil—selects the method invoked by a call based on the runtime types of any specified subset of the arguments [Steele Jr. 1990; Paepcke 1993; Shalit 1997; Feinberg et al. 1997; Chambers 1992; 1997]. A method that takes advantage of the multiple dispatch mechanism is called a *multimethod*. The generalization of receiver-based dispatch to multiple dispatch provides a number of advantages. For example, multimethods support safe covariant overriding in the face of subtype polymorphism, providing a natural solution to the binary method problem [Bruce et al. 1995; Castagna 1995]. More generally, multimethods are useful whenever multiple class hierarchies must cooperate to implement a method's functionality. For example, the code for handling an event in an event-based system depends on both which event occurs and which component is handling the event.

To motivate the addition of multiple dispatch in MultiJava, we consider the binary method problem in more detail. We revisit the `Shape` example from Fig. 1 and consider extending its `intersect` method to handle pairs of rectangles. First we describe the shortcomings of several approaches to performing this task in Java; then we illustrate MultiJava's solution.

2.1.1 *Binary Methods in Java.* There are several ways in Java by which one might attempt to implement an `intersect` method for pairs of rectangles.

2.1.1.1 *Static Overloading.* The first way one might attempt to add `Rectangle`'s `intersect` functionality in a Java program is shown in Fig. 2. Unfortunately, this approach does not provide the desired semantics. In particular, the new intersection method cannot safely override the original intersection method; the type of the non-receiver argument cannot safely be changed from `Shape` to the subtype `Rectangle` in the overriding method [Cardelli 1988]. Therefore, Java instead considers `Rectangle`'s `intersect` method to statically overload `Shape`'s method. Each method can be thought of as belonging to a distinct method family, as if they had completely different names. A *method family*[3] consists of a (possibly abstract) *top method*, which overrides no other methods, and all of the methods that override the top method. Java uses the name, number of arguments, and static argument types of a method to determine the family to which it belongs. In our example, the two `intersect` methods belong to different method families because they have

[3]In much of the literature on multiple dispatch languages, what we call a method family is referred to as a "generic function". We are using the term "method family" to avoid confusion with generic types in Java 1.5.

```
Rectangle rect1, rect2;
Shape shape1, shape2;

rect1 = new Rectangle( ... );
rect2 = new Rectangle( ... );

shape1 = rect1;
shape2 = rect2;

Shape i1 = rect1.intersect(rect2);
Shape i2 = rect1.intersect(shape2);
Shape i3 = shape1.intersect(rect2);
Shape i4 = shape1.intersect(shape2);
```

Fig. 3. Client code of the `intersect` method family

different static argument types.

Each method call expression in a Java program can invoke methods of only a single method family. The method family invoked by a method call is determined statically based on the call's name, number of arguments, and the static types of the actual arguments. The method invoked within that method family is determined at run time based on the dynamic type of the call's receiver object. For example, consider the client code in Fig. 3. Although the objects passed as arguments in the four `intersect` calls are identical, these calls do not all invoke the same method. In fact, only the first call will invoke `Rectangle`'s intersection method. The other three calls will invoke `Shape`'s intersection method, because the static types of these arguments cause Java to bind the calls to the method family introduced by `Shape`'s `intersect` method. Likewise, the first call is statically bound to the method family introduced by `Rectangle`'s `intersect` method. Therefore, new clients must choose statically which of the two method families they wish to invoke, and existing clients of `Shape` will never invoke the more efficient algorithm for two rectangles (unless they are modified to do so).

2.1.1.2 *Explicit Type Tests.* In Java, one can solve this problem by performing explicit runtime type tests with associated casts; we call this coding pattern a *typecase*. For example, one could implement the `Rectangle` intersection method as shown in Fig. 4.

This version of the `Rectangle` intersection method has the desired semantics. In addition, since it takes an argument of type `Shape`, this method can safely override `Shape`'s `intersect` method and is part of the same method family. All calls in the example client code of Fig. 3 will now invoke `Rectangle`'s `intersect` method. However, this code has several disadvantages. First, the programmer is explicitly coding the selection of the appropriate intersection algorithm, a process that can be tedious and error-prone. In addition, such code is not easily extensible. For example, suppose a `Triangle` subclass of `Shape` is added to the program. If special intersection behavior is required of a `Rectangle` and a `Triangle`, the above method must be modified to add the new case. Further, the case for such a new subclass must be carefully added in the appropriate place within the `if` expression, so that it will not be superseded by an earlier case. Finally, this solution loses static type safety. As a simple example, if the `instanceof` test in Fig. 4 accidentally tested

```
public class Rectangle extends Shape {
  ...
  public Shape intersect(Shape s) {
    if (s instanceof Rectangle) {
      Rectangle r = (Rectangle) s;
      /* efficient code for two Rectangles */
      ...
    } else {
      return super.intersect(s);
    }
  }
}
```

Fig. 4. An implementation of `Rectangle` using a typecase

whether `s` were an instance of `Shape` instead of `Rectangle`, the method would still typecheck properly but would cause a runtime `ClassCastException` to occur if a `Shape` instance were ever passed as the argument.

2.1.1.3 *Double Dispatch.* Another potential Java-based solution to the binary method problem is to use double dispatching [Ingalls 1986]. Fig. 5 shows an implementation of the `intersect` methods for `Shape` and `Rectangle` using double dispatching. With this technique, instead of using an explicit `instanceof` test to determine the runtime type of the argument `s`, as in the typecase solution, this information is obtained by performing a second call. This call is sent to the argument `s`, but with the name of the call encoding the dynamic class of the original receiver. Double dispatching reuses the language's built-in method dispatching mechanism, thereby retaining static type safety. However, double dispatching is even more tedious to implement by hand than typecases. Further, double dispatching requires at least the root class of the hierarchy to be modified whenever a new subclass is written. For example, the introduction of `Rectangle` in our example required `Shape` to be augmented with an `intersectRectangle` method. This modification is necessary even though there is no special `intersect` behavior desired for one shape and one rectangle.

2.1.2 *Multiple Dispatch in MultiJava.* MultiJava allows programmers to write multimethods, which are methods that can dynamically dispatch on other arguments in addition to the receiver object. Multimethods provide a simple solution to the binary method problem that does not suffer from the problems of the approaches described above. Multimethods also find more general applications; Section 5 demonstrates that they are useful whenever multiple arguments must cooperate to implement some functionality.

The syntax of our multimethod extension is specified in Fig. 6.[4] Using multimethods, the definition of the `Rectangle` class can be changed to the one shown

[4]The grammar given in Fig. 6 extends the Java syntax given in the first 17 chapters of *The Java Language Specification* [Gosling et al. 2000]. For standard Java nonterminals we just list the new productions for MultiJava and indicate the existence of the other productions with an ellipsis (. . .). Existing Java nonterminals bear superscript annotations giving the pertinent section numbers from the Java specification.

```
public class Shape {
  ...
  public Shape intersect(Shape s) {
    return s.intersectShape(this);
  }
  protected Shape intersectShape(Shape s) {
    ...
  }
  protected Shape intersectRectangle(Rectangle r) {
    /* no special code for one Shape and one Rectangle */
    return intersectShape(r);
  }
}
public class Rectangle extends Shape {
  ...
  public Shape intersect(Shape s) {
    return s.intersectRectangle(this);
  }
  protected Shape intersectRectangle(Rectangle r) {
    /* efficient code for two Rectangles */
    ...
  }
}
```

Fig. 5. Implementing binary methods using double-dispatching

FormalParameter$^{8.4.1}$:
Type$^{4.1}$ `@` *ReferenceType*$^{4.3}$ *VariableDeclaratorId*$^{8.3}$
...

Fig. 6. Syntax extensions for MultiJava multimethods

```
public class Rectangle extends Shape {
  ...
  public Shape intersect(Shape@Rectangle r) {
    /* efficient code for two Rectangles */
    ...
  }
}
```

Fig. 7. Multimethod version of `Rectangle`

in Fig. 7. This code is identical to the first solution attempt presented in Fig. 2, except that the type declaration of the formal parameter `r` is `Shape@Rectangle` instead of simply `Rectangle`. The "`Shape`" denotes the static type of the argument `r`.

```
public class Circle extends Shape {
  ...
  public Shape intersect( Shape s ) {
    ...
  }
  public Shape intersect( Shape@Rectangle r ) {
    ...
  }
  public Shape intersect( Shape@Circle c ) {
    ...
  }
}
```

Fig. 8. Another example of multimethods in MultiJava

Thus, `Rectangle`'s revised `intersect` method belongs to the same method family as `Shape`'s `intersect` method from Fig. 1—the name, number of arguments, and static argument types match. The "`@Rectangle`" indicates that, in addition to the receiver, we wish to dynamically dispatch on the formal parameter `r`.[5] We call `Rectangle` the *specializer* of parameter `r` and say that `r` is *specialized*. (We refer to a method without any specialized parameters as an *unspecialized* method.) As with standard Java, the receiver is always dispatched upon. So `Rectangle`'s `intersect` method will be invoked only if the dynamic class of the receiver is `Rectangle` or a subclass—as with Java—*and* the dynamic class of the argument `r` is `Rectangle` or a subclass. In all other cases, the `intersect` method from `Shape` will be invoked.

Any subset of a method's arguments can be specialized. A class can declare several methods with the same name and static argument types, with different argument specializers. For example, a `Circle` class could be defined with a selection of `intersect` methods as in Fig. 8. All these methods have static argument type `Shape`, so they all are in the same method family: the one introduced by the `intersect` method in the `Shape` class. However, they have different combinations of specializers, causing them to apply in different runtime circumstances.

MultiJava's multimethods have several advantages over the approaches described in Section 2.1.1. First, the dispatch on non-receiver arguments is expressed simply and declaratively. The various multimethods in a class can appear in any order, and the language does the work of choosing the appropriate method based on the runtime types of the arguments. Second, static type safety is retained. As will be described in Section 3, this includes checking for common errors including incompleteness and ambiguities among methods. Third, multimethods have all the properties of regular methods. For example, a later subclass `Square` of `Rectangle` can inherit `Rectangle`'s `intersect` multimethod, can optionally override that method, and can add new multimethods to handle other shapes specially.

2.1.3 *Multimethod Dispatch Semantics.* MultiJava's method invocation semantics, like Java's, can be broken down into two phases. The first, compile-time

[5] An alternative syntax would be to omit the `Shape@` part and just infer dynamic dispatch based on `Rectangle` being a subclass of `Shape` [Dutchyn et al. 2001]. However, this would conflict with static overloading in Java. We rejected this approach in favor of retaining backward compatibility.

selection of the appropriate method family, is the same in both languages. For a method call $E_0.I(E_1, \ldots, E_n)$, the method family being invoked is the unique method family named I that is in scope and is most appropriate for the static types of the E_i expressions [Gosling et al. 2000, pp. 346–355]. It is a compile-time error if there is not exactly one such method family. By using the same algorithm for compile-time selection of the appropriate method family, MultiJava retains Java's static overloading semantics.[6]

The second phase of method invocation is the dynamic selection of the appropriate method from the statically determined method family. MultiJava's semantics is a natural generalization of Java's second phase to handle dispatch on multiple arguments. Invocations to a method family whose methods do not use specializers will dispatch exactly as in Java. In MultiJava, dynamic dispatch for a method call $E_0.I(E_1, \ldots, E_n)$:

(1) evaluates each E_i to some value v_i,
(2) within the methods of the method family being invoked, finds the *most-specific applicable method*, M, for the argument tuple $(v_0, \ldots, v_n)$, and
(3) invokes M if it exists, or else signals an error.

The notion of a most-specific applicable method relies on a number of auxiliary definitions:

— First we define the natural subtyping relation for types. We say that a reference type D is a *direct subtype* of a reference type C if D `extends` or `implements` C. The *subtyping* relation is then the reflexive, transitive closure of the direct subtyping relationship on reference types, unioned with the identity relation on primitive types.

— Next we lift the notion of subtyping to tuples of types: *a type tuple* $(S_0, \ldots, S_n)$ *subtypes another type tuple* $(T_0, \ldots, T_n)$ if for all $0 \leq i \leq n$, it is the case that S_i subtypes T_i.

— Each argument tuple $(v_0, \ldots, v_n)$ has an associated *argument type tuple* $(D_0, \ldots, D_n)$, where for all $0 \leq i \leq n$, D_i is the dynamic type of v_i.

— Each method also has an associated *method type tuple* formed from its receiver type and the specializers of its parameters, or their static types if unspecialized. Formally, a method with n parameters, $I(P_1\ x_1, \ldots, P_n\ x_n)$, and a receiver type T_0 has the method type tuple $(T_0, T_1, \ldots, T_n)$, where for $i \in \{1..n\}$:
  - $T_i = P_i$ if parameter i is unspecialized, and
  - $T_i = D_i$ if $P_i = S_i$`@`D_i.

For example, the first method in Fig. 8 has the method type tuple (`Circle`, `Shape`), while the second method has the method type tuple (`Circle`, `Rectangle`).

Now we define an argument tuple's most-specific applicable method. This definition relies on two notions. First, we say that a method M is *applicable* to an argument tuple, $(v_0, \ldots, v_n)$, if the arguments' type tuple is a subtype of M's method type tuple. For example, for the method call

[6] In a language with multiple dispatch, static overloading becomes less necessary. Nevertheless, MultiJava must retain static overloading for backward compatibility with Java.

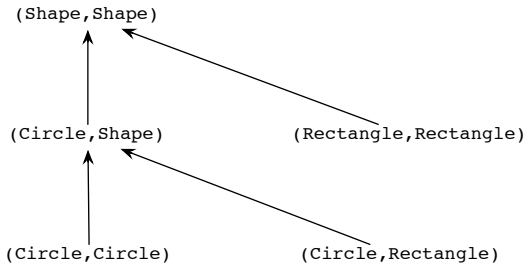

Fig. 9. Partial order on method type tuples for the `intersect` method family

```
new Rectangle( ... ).intersect(new Rectangle( ... )),
```

the argument type tuple is (`Rectangle`, `Rectangle`) and both the `intersect` method in `Shape` (Fig. 1) and the one in `Rectangle` (Fig. 7) are applicable. Second, we say that a method M_1 is *more specific* than a method M_2 if M_1's method type tuple is a subtype of M_2's. For example, the `intersect` method in `Rectangle` is more specific than the one in `Shape`.

Finally, the *most-specific applicable method* for a tuple of argument values is the unique applicable method that is more specific than all other applicable methods. If there are no applicable methods for a call, a *message-not-understood* error occurs. If there are applicable methods but no unique most-specific one, a *message-ambiguous* error occurs. (Static typechecking, discussed in Section 3, will detect and reject method families that could potentially cause either sort of runtime error.)

As an example we consider the `intersect` method family introduced by `Shape` in Fig. 1, and extended by `Rectangle` and `Circle` (in Figs. 7 and 8, respectively). Fig. 9 shows the partial order on this method family that is induced by the subtyping relationship on method type tuples. Each node in the figure is the type tuple for a method in the method family. Using this figure, we demonstrate MultiJava's dynamic dispatch semantics by considering the method call `shape1.intersect(shape2)` with varying runtime types for the arguments `shape1` and `shape2`.

— If both `shape1` and `shape2` are `Rectangle`s at run time, then the method with type tuple `(Rectangle,Rectangle)`, defined in the `Rectangle` class, is the most-specific applicable one.

— If `shape1` is a `Circle` and `shape2` is a `Shape`, then the method with type tuple `(Circle,Shape)`, the first method defined in `Circle`, is the most-specific applicable one.

— Finally, if `shape1` is a `Rectangle` and `shape2` is a `Circle`, then the top method, with type tuple `(Shape,Shape)`, is the most-specific applicable one.

2.1.3.1 *Symmetric vs. Asymmetric Dispatch.* Multiple dispatch is called *symmetric* if the rules for dynamic method lookup treat all dispatched arguments identically. *Asymmetric* multiple dispatch typically uses lexicographic ordering, where earlier arguments are treated as more important, to select between equally specific methods; a variant of this approach selects methods based partly on the textual

ordering of their declarations. Symmetric multiple dispatch is used in Cecil, Dylan, Kea [Mugridge et al. 1991], the $\lambda\&$-calculus [Castagna et al. 1995; Castagna 1997], $ML_{\leq}$ [Bourdoncle and Merz 1997], Tuple [Leavens and Millstein 1998], and Extensible ML [Millstein et al. 2002]. The asymmetric semantics is used in Common Lisp [Steele Jr. 1990; Paepcke 1993], Polyglot [Agrawal et al. 1991], Parasitic Methods [Boyland and Castagna 1997], and Half & Half [Baumgartner et al. 2002].

MultiJava employs the symmetric semantics. This is seen in the definition of subtyping for type tuples, which treats all argument positions uniformly. We think that symmetric multiple dispatch is more intuitive and less error-prone, reporting all possible ambiguities rather than silently resolving them in potentially unexpected ways.

2.1.3.2 *Handling* `null` *Arguments.* Method calls in Java have an inherent asymmetry in that `null` may be passed in a non-receiver argument position, but a `null` value in the receiver position results in the familiar `NullPointerException`. We considered throwing a similar exception in MultiJava if a `null` value appeared in any specialized argument position. This would arguably be the most symmetric treatment of the `null` value; however, this treatment would not be compatible with existing Java code that allows `null` values as arguments. For compatibility, MultiJava instead treats a `null` argument as having a specific runtime type: the static type of the corresponding parameter of the method family. For example, consider the following code fragment:

```
Shape shape1 = new Circle( ... );
Shape shape2 = null;
Shape result = shape1.intersect(shape2);
```

Because `shape2` is `null`, the argument type tuple for the `intersect` method call is `(Circle,Shape)`. Thus, the first `intersect` method of `Circle` is invoked. This semantics for `null` is consistent with what happens if a Java programmer uses runtime type tests to manually dispatch on argument types, since `null instanceof T` is false for all types `T`.

One consequence of MultiJava's treatment of `null` is that unspecialized methods must be concerned with possible `null` values for actual arguments. In his thesis, Clifton discusses an extension to MultiJava that would allow `null` as a specializer [2001, §6.1.6]. Such an extension would permit the declarative specification of methods to handle `null` arguments.

2.1.4 *Other Dispatch-Related Language Features.* Experience using multiple dispatch in MultiJava prompted us to add two additional dispatch-related language features. The first, resends, is analogous to `super` method calls in Java. The second feature, value dispatching, extends the declarative benefits of multimethods from the type domain to the value domain.

2.1.4.1 *Resends.* Inspired by a related construct in Cecil [Chambers 1997], MultiJava augments Java with a `resend` expression, which is similar to Java's `super` construct [Gosling et al. 2000, §15.12] but walks up the multimethod specificity ordering. An invocation of `resend` from a (multi)method invokes the unique most-specific (multi)method that is overridden by the resending method. (It is a compile-

time error if such a method does not exist.) For example, a call `this.resend(r)` from `Rectangle`'s `intersect` method in Fig. 7 would invoke `Shape`'s `intersect` method from Fig. 1. Referring to Fig. 9, the `resend` follows the arrow from the pair of `Rectangle`s to the pair of `Shape`s.

In this example, the `resend` has the same semantics as `super.intersect(r)`, but that is not always the case. A `super` send will always invoke a method whose receiver is a strict superclass of the current receiver (hence the `super` keyword appears in the receiver position of the call). But with a `resend` the invoked method may be less specific in some non-receiver argument position. For example, a `this.resend(r)` call from `Circle`'s second `intersect` method in Fig. 8 invokes the first method in that figure—following the arrow in Fig. 9 from (`Circle`,`Rectangle`) to (`Circle`,`Shape`). On the other hand, a `super.intersect(r)` call from the same method will invoke `Shape`'s `intersect` method.

To ensure that the unique most-specific method that is overridden by the resending one will be applicable to the `resend`'s arguments, MultiJava requires that the receiver of a `resend` be `this` and that the ith argument to the `resend` be the ith formal parameter of the enclosing method. Further, these formals must be declared `final` in the enclosing method, to guarantee that they are not modified before the call to `resend`. As a syntactic sugar, the target `this` may be omitted from a `resend` invocation. That syntax is also used for a `resend` from a static method.

An alternative design for resends would be to modify the semantics of `super` method calls instead of introducing a new keyword. An earlier version of MultiJava did exactly that. That design is still backward compatible with Java, differing from the semantics of `super` only where MultiJava's explicit specializers are present. However, that design has several drawbacks compared with the use of `resend`. First, some users were confused by the generalized semantics for `super`, expecting it to always invoke a method of the current receiver's superclass. Second, the original design could cause encapsulation problems when `super` was used with open classes to invoke a method of a different method family than the caller [Clifton 2001, p. 32]. The `resend` syntax solves this encapsulation problem by simply disallowing invocations of other method families. Finally, retaining Java's semantics for `super` aids the migration of a program from Java to MultiJava. For example, in the version of MultiJava with a modified `super` semantics, the target method of a `super` send in a Java method could unexpectedly change if the calling method were later converted into a set of multimethods.[7]

2.1.4.2 *Value Dispatching.* Multimethods are useful whenever a method family's behavior depends upon the particular runtime type of an argument. However, a method family's behavior sometimes depends upon an actual argument's *value* rather than its class. MultiJava's multimethods generalize naturally to support a common case of dispatch on values, in which the depended-upon values are compile-time constants.

Fig. 10 illustrates how value dispatching is used to compute the Fibonacci num-

[7]A third alternative, which we have not explored extensively, is to overload Java's `super()` syntax for invoking superclass constructors. This would avoid introducing a new keyword and, because the `super()` form can only appear in Java constructors, overloading it for resends would be unambiguous. On the other hand, we suspect the use of a new keyword is less confusing.

```
public class Fib {
  int fib(int@@0 i) { return 0; }
  int fib(int@@1 i) { return 1; }
  int fib(int i) { return fib(i-1) + fib(i-2); }
}
```

Fig. 10. Example of value dispatching

FormalParameter$^{8.4.1}$:
  Type$^{4.1}$ `@@` *ConstantExpression*$^{15.28}$ *VariableDeclaratorId*$^{8.3}$
  ...

Fig. 11. Syntax extensions for MultiJava value dispatching

bers. Value dispatch is denoted using `@@` instead of `@` on a specialized argument. Fig. 11 gives the formal syntax. The first method in Fig. 10 is only applicable dynamically if the argument `i` has the value 0, and similarly for the second method. The existing multimethod dispatch semantics generalizes seamlessly to handle value dispatching by viewing each dispatched-upon value as a singleton concrete subclass of its associated type. For example, the first `fib` method overrides the third one because 0 is a "subclass" of `int`. Value dispatching is supported for the literals of all of Java's primitive types, as well as for literals of type `java.lang.String`. Further, any compile-time constant expression as defined by Java can be used as a value specializer, in addition to simple literals. Examples that make use of this ability are described in Section 5.

Value dispatching as shown in the example is similar to Java's existing `switch` statement, but value dispatching has a number of advantages. First, any subset of a method's arguments can employ value dispatching, and a method can employ value dispatching on some arguments and ordinary class dispatching on others. Second, value dispatching is supported for all of Java's primitive types, instead of just the integral ones, as well as for `java.lang.String`. Finally, methods that employ value dispatching are inherited by subclasses, providing more extensibility than `switch`. For example, a subclass of `Fib` can inherit some of the `fib` methods, override others, and add new multimethods handling other interesting integer values.

2.2 Open Classes

In addition to multimethods, MultiJava also supports open classes. An *open class* is one to which new methods can be added without editing the class directly [Chambers 1998; Millstein and Chambers 2002]. An open class allows a client to easily customize the class's interface to the needs of an application, without modifying existing code. Open classes can be used to organize "cross-cutting" operations separately from the classes to which they belong, a key feature of subject-oriented and aspect-oriented programming [Harrison and Ossher 1993; Kiczales et al. 1997]. With open classes, object-oriented languages can support the addition of both new subclasses and new methods to existing classes, solving the augmenting method

```
public class CircumferenceShape extends Shape {
  public double circumference() {
    return accumDistance( borderPoints() );
  }
}
```

Fig. 12. Adding new operations by subclassing

problem described in Section 1.1.2.

Similar to Section 2.1, we motivate the inclusion of open classes in MultiJava by first considering Java-based solutions to the augmenting method problem.

2.2.1 *Approaches to Solving the Augmenting Method Problem in Java.* One solution to the augmenting method problem is to add the new operation by writing new subclasses for each of the existing classes. For example, Fig. 12 illustrates how a `circumference` method is added to `Shape`. There are several drawbacks of this approach. First, the new method does not update any existing `Shape` instances, for example from a persistent store. Second, in general existing clients of `Shape` must still be modified: any code that constructs `Shape` instances must be updated to construct `CircumferenceShape` instances. Third, if `CircumferenceShape` instances are passed to and later returned from existing code, their static type upon return will be `Shape` (unless the existing code is modified), requiring runtime casts to regain access to the `circumference` operation. Finally, this technique is tedious and awkward when an entire class hierarchy must be updated with a new operation. For example, we must also declare a `CircumferenceRectangle` subclass of `Rectangle`. `CircumferenceRectangle` must subclass from `CircumferenceShape` in addition to `Rectangle`, in order for the new class's `circumference` method to override that of `CircumferenceShape`. Therefore, implementing `CircumferenceRectangle` requires multiple inheritance of classes, which is not available in Java.

A second approach is to use the Visitor design pattern [Gamma et al. 1995, pp. 331–344], which directly addresses the problem of adding new functionality to existing classes in a modular way. The basic idea is to reify each operation into a class, thereby allowing operations to be structured in their own hierarchy. For example, consider the version of the `Shape` class hierarchy in Fig. 13; here the classes are augmented with an `accept` method according to the Visitor pattern. Operations on shapes are structured in their own class hierarchy, each operation becoming a subclass of an abstract `ShapeVisitor` class as shown in Fig. 14. The client of an operation on shapes invokes the `accept` method of a shape, passing a `ShapeVisitor` instance representing the operation to perform:

```
someShape.accept(new CircumferenceVisitor( ... ));
```

The `accept` method of each kind of shape uses double dispatching to invoke the method of the visitor that is appropriate for that shape.

The main advantage of the Visitor pattern is that new operations can be added modularly, without needing to edit any of the `Shape` subclasses: the programmer simply defines a new `ShapeVisitor` subclass containing methods for visiting each class in the `Shape` hierarchy. However, use of the Visitor pattern brings several

```
public class Shape {
  ...
  public void accept( ShapeVisitor v ) {
    v.visitShape( this );
  }
}

public class Rectangle extends Shape {
  ...
  public void accept( ShapeVisitor v ) {
    v.visitRectangle( this );
  }
}
```

Fig. 13. Shape hierarchy with Visitor infrastructure

```
public abstract class ShapeVisitor {
  ...
  public abstract void visitShape( Shape s );
  public abstract void visitRectangle( Rectangle r );
  /* abstract methods for other Shape subclasses */
  ...
}

public class CircumferenceVisitor extends ShapeVisitor {
  private double result;
  public double getResult() { return result; }
  ...
  public void visitShape( Shape s ) {
    result = accumDistance( s.borderPoints() );
  }
  public void visitRectangle( Rectangle s ) {
    result = 2.0 * (r.height() + r.width());
  }
}
```

Fig. 14. Operation on `Shape`s implemented with Visitor

drawbacks, including the following, listed in order of increasing importance:

— The stylized double-dispatching code is tedious to write and prone to error.

— The need for the Visitor pattern must be anticipated ahead of time, when the `Shape` class is implemented. For example, had the `Shape` hierarchy not been written with an `accept` method, which allows visits from the `ShapeVisitor` hierarchy, it would not have been possible to add the circumference functionality in a modular way.

— Even with the `accept` method included, only visitors that require no additional arguments and that return no results can be programmed in a natural way; for example, `CircumferenceVisitor` must use the `result` field and the `getResult` accessor method to store and return the new operation's result.

— Although the Visitor pattern allows the addition of new operations modularly, in so doing it gives up the ability to add new subclasses to existing `Shape` classes in a modular way. For example, if a `Circle` subclass were introduced, the `ShapeVisitor` class and all subclasses would have to be modified to contain a `visitCircle` method. Thus, Visitor trades the non-modularity of the object-oriented approach for the non-modularity of the procedural approach. Proposals have been advanced for dealing with this well-known limitation [Krishnamurthi et al. 1998; Martin 1998; Palsberg and Jay 1998; Nordberg 1998; Vlissides 1999; Zenger and Odersky 2001; Grothoff 2003; Torgersen 2004]. However, most of these proposals suffer from additional complexity (in the form of hand-coded typecases, more complex class hierarchies, and factory methods) that make them even more difficult and error-prone to use. Further, some of the proposals require a loss of static type safety in the form of runtime casts or reflection in order to resolve Visitor's limitation. All of the proposals require advance planning by the developer of the code to be extended.

2.2.2 *Open Classes in MultiJava.* The open class feature of MultiJava allows a programmer to add new methods to existing classes without modifying existing code and without breaking the encapsulation properties of Java. Contrary to the Visitor pattern, it does this in a way that still allows new subclasses to be introduced modularly. Thus MultiJava's open classes solve the augmenting method problem.

2.2.2.1 *Declaring and Invoking External Methods.* The key new language feature involved in open classes is the *external method declaration*, whose syntax is specified in Fig. 15. Using external methods, the functionality of the circumference-calculating visitor from Fig. 14 can be written as in Fig. 16. The two external methods in the figure belong to a new method family added to the shape hierarchy. We call that method family *external* because its top method (the first one in the figure) is an external method. Method families whose top method is not external, including regular Java method families, are called *internal*.

As in Java methods, the bodies of external methods may use the keyword "`this`" to reference the receiver argument. Similarly, field references and method calls can implicitly target the receiver, for example in the calls to `width()` and `height()` in Fig. 16. Finally, external methods may also be multimethods, by employing MultiJava's syntax for declaring argument specializers.

Clients invoke external method families exactly as they would the class's original methods. For example, the `circumference` method of a `Shape` instance, `someShape`, is invoked by `someShape.circumference()`. This is allowed even if the instance referred to by `someShape` was retrieved from a persistent database, or was created by code that did not have access to the `circumference` methods. Code can create and manipulate instances of classes without being aware of all external method families that may have been added to the classes; only code wishing to invoke or extend a particular external method family needs to be aware of its declaration.

2.2.2.2 *Scoping of External Method Families.* To invoke or override an external method family, client code first imports it using an extension of Java's existing import mechanism. For example,

(a) *ExternalMethodDeclaration*:
ExternalMethodHeader *MethodBody*$^{8.4.5}$

ExternalMethodHeader:
MethodModifiers$^{8.4.3}_{opt}$ *ResultType*$^{8.4}$ *ExternalMethodDeclarator* *Throws*$^{8.4.4}_{opt}$

ExternalMethodDeclarator:
ClassOrInterfaceType$^{4.3}$ `.` *Identifier*$^{3.8}$ `(` *FormalParameterList*$^{8.4.1}_{opt}$ `)`

(b) *CompilationUnit*$^{7.3}$:
PackageDeclaration$^{7.4.1}_{opt}$ *ImportDeclarations*$^{7.5}_{opt}$ *TopLevelDeclarations*$_{opt}$

TopLevelDeclarations:
TopLevelDeclaration
TopLevelDeclarations *TopLevelDeclaration*

TopLevelDeclaration:
TypeDeclaration$^{7.6}$
ExternalMethodDeclaration

Fig. 15. (a) Syntax for MultiJava external methods, and (b) syntax extensions to declare external methods outside of class declarations

```
/* In file "circumference.java" */
package examples;

public double Shape.circumference() {
  return accumDistance(borderPoints());
}

public double Rectangle.circumference() {
  return 2.0 * (width() + height());
}
```

Fig. 16. Circumference-calculating method family using external methods

```
import examples.circumference;
```

will import the method family `circumference` from the package `examples`. Similarly

```
import examples.*;
```

will implicitly import all the compilation units in the package `examples`, which will make all accessible (e.g., public) types and method families in that package available for use. Each compilation unit implicitly imports all the method families in its package.

We call the set of methods and fields in a class the *signature* of that class. The *apparent signature* of a class for a given client is the set of method families and fields available to that client. The explicit importation of external method families enables client code to manage the apparent signatures of the classes it manipulates. Only clients that import the `circumference` method family will see the `circumference` operation in the apparent signature of `Shape`. Other clients will not have

```
import examples.circumference;

public class Parallelogram extends Shape {
  ...

  /* Implements circumference method family for Parallelograms */
  public double circumference() {
    return 2.0 * (base() + side());
  }
}
```

Fig. 17. Example of modularly adding a new subtype to an existing type hierarchy

their apparent signatures for `Shape` polluted with this method family. Furthermore, a compilation unit that did not import the existing `circumference` method family could declare its own `circumference` method family without conflict. (As in Java, a MultiJava compiler will signal a compile-time error if multiple applicable method families are in scope at a call site.) The scoping of external general functions is one of the ways that MultiJava's open classes are more modular than open classes in AspectJ [AspectJ Team 2004].

Java allows at most one public type (class or interface) declaration in a compilation unit.[8] This concession allows an implementation to find the file containing the source code or bytecode for a type based on its name. In MultiJava we extend this restriction in a natural way: each file may contain either one public type with associated methods, or a collection of external methods of the same name. Typically these methods will all belong to a single method family, as in Fig. 16, but we also allow the methods to belong to multiple, statically overloaded method families.

2.2.2.3 *Inheritance of External Methods.* Unlike the Visitor design pattern, open classes still allow a new subclass of `Shape` to be added without changing any existing code. This capability arises from two important features of inheritance in MultiJava. We use the `Parallelogram` class of Fig. 17 to explain these features.

First, in MultiJava a subclass can override any method in the apparent signature of its superclass. That is, a new subclass can

—import an external method family that augments the superclass, and

—add an overriding method to the method family.

The compilation unit for `Parallelogram` does this by importing `circumference` and declaring its own `circumference` method.

The second inheritance feature for external method families applies to clients of a new subclass. A client of a new subclass can

—import an external method family that augments the superclass and

—invoke that method family on a subclass instance, regardless of whether the method family was visible from the subclass's declaration.

For example, suppose that a client program imported the `area` method family from

[8] Java's restriction is actually somewhat more complex to account for its default access modifier, which gives access to all other classes in the package [Gosling et al. 2000, §7.6]

```
/* In file "area.java" */

public double Shape.area {
  return sumTriangles( sort( borderPoints() ) );
}

public double Rectangle.area {
  return width() * height();
}
```

Fig. 18. Example used to show method inheritance for open classes

Fig. 18. Even though the `area` method was not in the apparent signature of `Shape` from `Parallelogram`'s perspective, the client can still execute the following code:

```
Parallelogram par = new Parallelogram( ... );
double area = par.area();
```

Because the client imports the `area` method family and thus adds it to `Shape`'s apparent signature, `Parallelogram` implicity inherits the `Shape.area()` method from Fig. 18, so that method is invoked by the above code.

2.2.2.4 *Encapsulation.* MultiJava retains the same encapsulation properties as Java [Gosling et al. 2000, §6.6]. An external method may access:

—public members of its receiver class, and

—non-private members of its receiver class if the external method is in the same package as that class.

All other access to receiver class members is prohibited. Therefore, an external method does not typically have access to the private members of its receiver class. This does limit the expressiveness of external methods as compared with the ordinary methods of a class, but it allows us to retain Java's strong encapsulation properties. Providing private access to the receiver from an external method would allow any client to access a class's implementation simply by declaring an external method for the class.

An external method may be declared with any one of the Java access modifiers. For example, a helper method for a public external method may be declared private and included in the same compilation unit as the public method. An external method's modifier is defined relative to the current compilation unit. For example:

— A private external method may only be invoked or overridden from within the compilation unit in which it is declared.

— A protected external method may only be invoked or overridden by a class in the same package in which the method is declared or by a class that is a subtype of the method's receiver.

2.3 Why Both?

It might seem that open classes are unnecessary given that MultiJava includes multimethods. For example, one might construct a method family for calculating the circumference of shapes using multimethods instead of open classes, as in Fig. 19. With this definition a client can find the circumference of a shape with

```
public class CircumferenceCalculator {
  public double circumferenceFor( Shape s ) { ...}
  public double circumferenceFor( Shape@Rectangle r ) { ...}
  public double circumferenceFor( Shape@Circle c ) { ...}
}
```

Fig. 19. noninvasive visitor using multimethods

```
double c = new CircumferenceCalculator().
                          circumferenceFor( someShape );
```

effectively adding an operation to the `Shape` hierarchy without using open classes.

We say that a method family coded in this fashion is a *noninvasive visitor* [Millstein 2003]. The term "visitor" comes from the obvious similarity of this code to the usual Visitor pattern. The code is noninvasive because it does not require any changes to the shape classes (for example, to declare `accept` methods).

There are two problems with the noninvasive visitor approach, which are not shared by MultiJava's open classes. First, the invocation syntax for the `circumferenceFor` method family is different from the invocation syntax for method families declared inside the original shape classes. The second problem is more onerous. As with the Visitor pattern, noninvasive visitors lose the ability to add new `Shape` subclasses in a modular way. New subclasses would require either a non-modular editing of the `CircumferenceCalculator` class or the creation of a subclass of `CircumferenceCalculator`, which has the same problems as described for the subclassing solution to the augmenting method problem (see Section 2.2.1). Therefore, noninvasive visitors do not fully solve the augmenting method problem.

3. STATIC TYPECHECKING

MultiJava extends Java's static type system to accommodate multimethods and external methods. This involves enhancements to the typechecking of method families. Importantly, MultiJava's type system remains *modular*, obeying Java's file-by-file typechecking strategy. After more specifically defining the notion of modular typechecking, we describe *client-* and *implementation-side typechecking* [Chambers and Leavens 1995] of method families in MultiJava.

3.1 Modular Typechecking

We say that a type system is *modular* if the typechecking of each compilation unit obeys the following two properties.

— First, each compilation unit can be typechecked given only the interfaces of other compilation units, without requiring knowledge of their implementation details. Informally, in Java a type's interface information consists of the names of its superclass and superinterfaces, the types of its accessible fields, and the headers, but not bodies, of its accessible methods. Formal definitions have been provided by others [Drossopoulou et al. 1999; Ancona et al. 2002]. For MultiJava, the interface information of a compilation unit also includes the headers, but not bodies, of the accessible external methods.

— Second, each compilation unit U can be typechecked given access only to the interface information of compilation units that U explicitly *depends upon*. These are

the interfaces that define types and method families referenced by U, as well as the interfaces that define types and method families referenced by U's depended-upon interfaces (recursively). Intuitively, the depended-upon interfaces are those whose compilation units must exist in the program, in order for U to be well formed. For example, `Rectangle` in Fig. 7 depends upon the interface of `Shape` in Fig. 1, because `Rectangle` refers to `Shape`. However, `Rectangle` does not depend upon `Circle` in Fig. 8 and so should not have to access `Circle`'s interface during typechecking. Indeed, `Circle` may not even exist at the time that `Rectangle` is typechecked. We say that the classes that are in interfaces upon which U depends are *visible* during U's typechecking. We use a similar definition for the visibility of methods and method families.

3.2 Client-side Typechecking

Client-side typechecking ensures the type correctness of each method call. MultiJava performs the same client-side typechecking as does Java, and the check is naturally modular. Given a method call of the form $E_0.I(E_1, \ldots, E_n)$, the receiver and arguments are typechecked, and their static types are used to find the unique method family being invoked (among the many that may be statically overloaded). If such a method family is found, then the method call is well typed and is assigned the return type of the method family. Otherwise, an error occurs.

3.3 Implementation-side Typechecking

Implementation-side typechecking ensures the type correctness of the set of methods belonging to each method family. (We ignore value dispatching here; it is addressed in Section 3.4 below.)

3.3.1 *Local Checks.* First, there are checks on individual methods. As in Java, a method must have modifiers that are compatible with those of the methods it overrides, and the method's body must match the declared return type. MultiJava uses additional information in performing the latter check: when typechecking a method body in MultiJava, we can safely assume that the arguments have the types of their specializers, rather than simply their static types. Indeed, this is a major part of the convenience of MultiJava. MultiJava also checks that each specializer is a proper subtype of its static type. Because all of these checks are local to individual methods, they are naturally modular.

3.3.2 *Checks that Method Families are Properly Implemented.* The remaining part of implementation-side typechecking is responsible for ensuring that each method family is *properly implemented*: for each *valid type tuple* to the method family there must be a most-specific applicable method. A tuple of classes $(C_0, \ldots, C_n)$ is a valid type tuple for a method family with argument types (including the receiver) $(T_0, \ldots, T_n)$ if each C_i is a concrete class that is a subtype of the associated T_i. If a method family is properly implemented, then method lookup on that family always succeeds at run time, with no message-not-understood or message-ambiguous errors. For example, consider checking that the `intersect` method family is properly implemented in the context of Figs. 1, 7, and 8. There are nine valid type tuples to check — all possible pairs of `Shape`s, `Rectangle`s, and `Circle`s. Each of the nine argument tuples has a most-specific applicable method, so

the check succeeds.

Java's type system already ensures that each method family is properly implemented, in the absence of external methods and multimethods. For example, if an abstract class C declares an abstract method m, then a concrete subclass of C must provide a concrete overriding method of m, or else the typechecker signals an error. Without such a concrete method, a `NoSuchMethodException` exception (the equivalent of our message-not-understood error) could occur at run time. Java's check that method families are properly implemented is completely modular: typechecking on each class ensures that the class has a most-specific method for each method family to which it can be passed as a receiver argument. On the other hand, MultiJava's check as described so far is *global*: checking a method family f requires access to all of f's methods and all valid type tuples.

To make MultiJava's check modular, we divide it into checks on individual compilation units. The typechecking on a compilation unit U must ensure that all *visible* method families are properly implemented when considering *visible* types. That is, all visible valid type tuples must have a visible most-specific applicable method to invoke. Consider again checking that `intersect` is properly implemented in Figs. 1, 7, and 8. Checks on the `Shape` class ensure that `intersect` has a most-specific applicable method for a pair of `Shape`s, as `Shape` is the only visible class. Checks on the `Rectangle` class ensure that `intersect` properly handles all pairs of `Shape`s and `Rectangle`s, but not `Circle`s; the `Circle` class is not visible. Checks on the `Circle` class ensure that `intersect` properly handles all pairs of `Shape`s, `Rectangle`s, and `Circle`s.

Unfortunately, this natural scheme for making the check modular is insufficient: there can be message-not-understood and message-ambiguous errors that elude modular static detection [Cook 1991]. To address this problem, we impose some requirements that, together with the modular check described above, are enough to ensure that each method family is in fact properly implemented. These requirements are adapted from our previous theoretical work on modular typechecking for multimethods and open classes [Millstein and Chambers 2002; Millstein et al. 2002; Millstein 2003]. Below we briefly illustrate the challenges for modularly ensuring that method families are properly implemented and show how the additional requirements resolve the problems. More details on the requirements can be found in the earlier papers, including a type soundness proof that validates their sufficiency. The requirements are partitioned into those that ensure a method family is *complete* (i.e., no message-not-understood errors) and those that ensure a method family is *unambiguous* (i.e., no message-ambiguous errors); we discuss each kind in turn.

3.3.2.1 *Ensuring Completeness.* Internal and external method families are subject to different completeness requirements. Fig. 20 illustrates a completeness problem for internal method families. Unlike our previous examples, in this figure `Shape` is declared abstract. Checks that `intersect` is properly implemented from `Shape`'s compilation unit succeed vacuously: because `Shape` is abstract, there are no valid visible type tuples to check. The checks from `Rectangle`'s compilation unit succeed because the single valid visible tuple, (`Rectangle`, `Rectangle`), has a most-specific applicable method, and similarly for the checks from `Circle`'s compilation unit.

```
/* compilation unit "Shape" */
public abstract class Shape {
  ...
  public abstract Shape intersect(Shape s);
}
/* compilation unit "Rectangle" */
public class Rectangle extends Shape {
  ...
  public Shape intersect( Shape@Rectangle r ) {
    ...
  }
}
/* compilation unit "Circle" */
public class Circle extends Shape {
  ...
  public Shape intersect( Shape@Circle c ) {
    ...
  }
}
```

Fig. 20. A completeness problem for multimethods

However, at run time a message-not-understood error will occur if `intersect` is invoked on a pair of one `Rectangle` and one `Circle`, or vice versa.

To solve the problem, we require `Rectangle` to declare an unspecialized method (i.e., a regular Java method) for `intersect`. This method ensures that `Rectangle` implements `intersect` for any potentially unseen shape arguments, thereby handling the incompleteness for a pair of one `Rectangle` and one `Circle`. Similarly, `Circle` must declare an unspecialized method for `intersect`, thereby handling the incompleteness for a pair of one `Circle` and one `Rectangle`. In general, the requirement is as follows:

> **R1.** If a concrete class C declares or inherits an internal method family f, then C must declare or inherit a concrete unspecialized method for f. Also, a method with specialized arguments (a multimethod) may not be declared `abstract`.

Requiring a concrete unspecialized method is no extra burden on Java programmers, as it is exactly the requirement that Java already enforces to ensure completeness of internal method families. In our example, Java would require both `Rectangle` and `Circle` to contain a concrete `intersect` method, because the one in `Shape` is abstract. Requirement R1 also forbids multimethods from being abstract. This is necessary to ensure that the required unspecialized method will be applicable for all valid type tuples.[9]

Fig. 21 illustrates a completeness problem for external method families. Suppose again that `Shape` is declared abstract. Because the compilation unit in Fig. 21 only depends upon the compilation units of `Shape` and `Rectangle`, `circumference`

[9]In Java, a concrete method is not applicable to a method call if there is an overriding abstract method that is also applicable.

```
/* compilation unit "circumference" */
package examples;

public abstract double Shape.circumference(); /* violates R2 */

public double Rectangle.circumference() {
  return 2.0 * (width() + height());
}
```

Fig. 21. A completeness problem for open classes

appears complete: the single valid visible type tuple, (`Rectangle`), has a most-specific applicable method. However, if `Circle` from Fig. 8 is also linked into the program, we will get a message-not-understood error at run time if `circumference` is ever invoked on a `Circle` instance.

Unlike the situation of internal method families, `circumference` is not visible from `Circle`'s compilation unit (indeed, `circumference` may not even have been written yet), so `Circle` cannot be required to declare a `circumference` method. Instead, we require the compilation unit declaring `circumference` to declare a concrete *default* method, which is a method that accepts any argument tuple of the appropriate type. The requirement can be satisfied in Fig. 21 by making the first method concrete; that method then handles the unseen `Circle` class. In general, the requirement is as follows:

> **R2.** The compilation unit that introduces an external method family f must declare a concrete default method for f. Also, an external method may not be declared `abstract`.

Requirement R2 has the effect of treating abstract classes as if they were concrete for the purposes of external method families. The requirement therefore does limit the expressiveness of external method families, sometimes requiring default methods that are unnecessary or unnatural. Of course, the requirement still allows MultiJava to be strictly more expressive than Java, which lacks external methods altogether. Analogous with requirement R1, external methods are forbidden from being abstract to ensure that the required default method is always applicable.

3.3.2.2 *Ensuring the Absence of Ambiguity.* Fig. 22 illustrates an ambiguity problem for multimethods. Both methods in the figure belong to the `intersect` method family defined by the top method in Fig. 1. Because neither the "Rectangle" nor "intersect" compilation unit depends upon the other, each unit successfully typechecks. In particular, `intersect` appears to be properly implemented from each compilation unit. However, at run time a message-ambiguous error will occur if `intersect` is invoked on a pair of one `Rectangle` and one `Circle`: both methods in the figure are applicable, but neither is more specific than the other.

To handle this problem we impose a requirement that disallows the second `intersect` method in Fig. 22 from being written, thereby resolving the potential ambiguity. The requirement is as follows:

> **R3.** An external method must be declared in the same compilation unit as any methods that it overrides, and those methods must also be external.

```
/* compilation unit "Shape" as in Fig. 1 */

/* compilation unit "Rectangle" */
public class Rectangle extends Shape {
  ...
  public Shape intersect( Shape s ) {
    ...
  }
}

/* compilation unit "intersect" */
public Shape Shape.intersect( Shape@Circle c) { /* violates R3 */
  ...
}
```

Fig. 22. An ambiguity problem

```
/* compilation unit "circumference" */
package examples;

public double Shape.circumference(){
  ...
}

public double Rectangle.circumference() { /* violates R4 */
  ...
}

public double Rhombus.circumference() { /* violates R4 */
  ...
}

/* compilation unit "Square" */
public class Square implements Rectangle, Rhombus {
  ...
}
```

Fig. 23. An ambiguity problem from multiple inheritance

The second `intersect` method in our example violates the requirement, because `intersect`'s top method is declared in `Shape`'s compilation unit.

Requirement R3 is no burden on the traditional Java style, since it only affects external methods. Ordinary Java-style overriding is allowed, as illustrated by the first `intersect` method in Fig. 22, and regular methods may still belong to external method families, as in Fig. 17. The requirement does restrict the external methods that may be written. First, an external method must belong to an external method family. Second, all external methods of a given method family must be declared in the same compilation unit. The requirement ensures that if two methods are in compilation units such that neither unit depends upon the other, then the two methods will be applicable to disjoint sets of valid type tuples. Therefore, the two methods cannot be ambiguous with one another.

Finally, Fig. 23 illustrates an ambiguity problem in the presence of Java's interfaces, which support multiple inheritance. For this example, suppose that `Shape` is an interface, as are its subinterfaces `Rectangle` and `Rhombus`. Because neither of the two compilation units in the figure depends upon the other, the ambiguity of `circumference` for `Square` eludes modular static detection. To handle this problem, we disallow dynamic dispatch on interfaces, either as the receiver or as an argument:

> **R4.** If an external method's receiver is an interface, then the method must be its method family's top method. Also, an interface may not be used as a specializer.

In Fig. 23, the first `circumference` method is allowed by the requirement because it is the top method, while the other two methods are disallowed. The top method may safely have an interface for a receiver because in that case the receiver is not being dynamically dispatched upon: the top method is only invoked if there is no applicable, more-specific method. Requirement R4 naturally generalizes Java's requirement that an interface contain no concrete methods, which prevents interface types from affecting dynamic dispatch.[10]

3.4 Implementation-side Typechecking for Value Dispatching

We generalize implementation-side typechecking to support value dispatching. The local checks on individual methods, as described in Section 3.3.1, are augmented to check that each value specializer is a compile-time constant expression that has the argument's associated static type. The checks that a method family is properly implemented are augmented by treating each value used as a specializer as a singleton concrete subclass of its associated type. For example, during checks on `fib` in Fig. 10, the values `0` and `1` are checked to have a most-specific applicable method. This checking ensures, among other things, that an ambiguity will be signaled statically if multiple `fib` methods dispatch on `0`.

4. MODULAR COMPILATION

The compilation strategy for MultiJava generates standard Java bytecode and retains the modular compilation and efficient single dispatch of existing Java code. Additional runtime cost for MultiJava's new features is incurred only where they are used; code that does not make use of multiple dispatch or open classes compiles and runs exactly as in Java.

We have implemented our modular compilation strategy (as well as the modular typechecking strategy described in the previous section) in `mjc`, a compiler for MultiJava. The `mjc` compiler is built as an extension to the Kopi compiler, an open-source Java compiler [Kopi 2004].

The next subsection describes the compilation strategy for multimethods, focusing on just internal method families. Section 4.2 describes the translation and invocation of external method families. Section 4.3 describes the compilation of

[10]The original design of MultiJava did not permit any external methods whose receiver was of an interface type. This restriction was relaxed to that of **R4** based on user feedback as discussed in Section 5.2.1.

super calls and resends. Finally, Section 4.4 discusses some miscellaneous compilation issues. Although `mjc` outputs Java bytecode, to simplify discussion we will describe compilation as if translating to Java source code.

4.1 Compiling Multimethods

Multimethods in MultiJava are compiled in the style of encapsulated multimethods [Bruce et al. 1995], though the complexity of this style is hidden from the programmer.

All the multimethods of a given internal method family within a single class are grouped in a *dispatcher method*. Consider the set of `intersect` methods in Fig. 24a. For such a set of multimethods, the MultiJava compiler produces a dispatcher method, as shown in Fig. 24b, that selects the appropriate multimethod at run time. The dispatcher method internally does the necessary checks on the specialized arguments, using cascaded sequences of `instanceof` tests (or equality comparisons, for value dispatching). The multimethod bodies are translated into a set of private methods.

For the set of multimethods compiled into a dispatcher method, the dynamic dispatch tests are ordered to ensure that the most-specific multimethod is found. If one of the multimethods in the set is applicable to some argument tuple, then the typechecking restrictions ensure that there will always be a single most-specific check which succeeds. Moreover, the multimethod body selected by this check will be more specific than any applicable superclass method: the subclass method is always more specific in the receiver position than any superclass method.

If every multimethod compiled into a dispatcher method has a specializer on some argument position, then it is possible that none of the checks will match the runtime arguments. In this case, a final clause passes the dispatch on to the superclass, as shown in Fig. 24b. Eventually a class must be reached that includes an unspecialized method, as required by typechecking requirement R1.

Compiling Java single dispatch methods is just a special case of these rules. Such a method does not dispatch on any arguments and has no other local multimethods overriding it, and so its body performs no runtime type dispatch on any arguments; it reduces to just the original method body.

An invocation of an internal method family is compiled exactly as in Java. Clients are insensitive to whether or not the invoked method family performs any multiple dispatch. Thus, the set of arguments on which a method dispatches can be changed without needing to retypecheck or recompile clients.

There is no efficiency penalty for Java code compiled with the MultiJava compiler. Only methods that dispatch on multiple arguments get compiled with typecases. A Java program would likely use typecases whenever a MultiJava program would use multimethods anyway. If a Java program used double dispatching to simulate multimethods, then it might be possible to generate more efficient code than MultiJava (two constant-time dispatches), but double dispatching in Java sacrifices the ability to add new subclasses modularly.

4.2 Compiling External Method Families

The external method families introduced by MultiJava's open class mechanism are compiled using *anchor classes*, generated classes that represent the external method

(a)
```
/* Implements Square in MultiJava */
public class Square extends Rectangle {
  ...
  public Shape intersect( Shape@Rectangle r ) {
    /* method 1 body */
    ...
  }
  public Shape intersect( Shape@Square s ) {
    /* method 2 body */
    ...
  }
}
```

(b)
```
/* Translation of Square to Java */
public class Square extends Rectangle {
  ...
  /* Generated "dispatcher method" */
  public Shape intersect( Shape sh ) {
    if (sh instanceof Square) {
      return intersect$body2( (Square) sh );
    } else if (sh instanceof Rectangle) {
      return intersect$body1( (Rectangle) sh );
    } else {
      return super.intersect( sh );
    }
  }
  private Shape intersect$body1( Rectangle r ) {
    /* method 1 body */
    ...
  }
  private Shape intersect$body2( Square s ) {
    /* method 2 body */
    ...
  }
}
```

Fig. 24. Example showing (a) multimethods in MultiJava, and (b) their translation to Java

family. The anchor classes allow external methods to be compiled apart from the types that they augment. An anchor class has a single static field, `function`, that contains a *dispatcher object*. During an invocation of the external method family, the dispatcher object is responsible for invoking one of the method family's methods based on MultiJava's dynamic dispatch semantics. The dispatcher object is a Java implementation of a first-class function; it allows the method family's methods to be stored in a field. We will see below how this helps in implementing overriding methods.

As an example, Fig. 25a introduces the `rotate` external method family. Fig. 25b shows the anchor class generated by our compiler. The anchor class's access level is based on the declared access modifier of the external method family. The anchor class declares nested types representing the dispatcher object and its interface. As

(a)
```
/* Implements rotate in MultiJava */
public Shape Shape.rotate(float a) { /* method 1 body */  ... }
public Shape Rectangle.rotate(float a) { /* method 2 body */  ... }
public Shape Square.rotate(float a) { /* method 3 body */  ... }
```

(b)
```
/* Translation of rotate to Java */
public class rotate$anchor {

  public static signature function;
  static {
    synchronized (rotate$anchor.class) {
      function = new dispatcher();
    }
  }

  public interface signature {
    Shape apply(Shape this$, float a);
  }

  private static class dispatcher implements signature {
    public Shape apply( Shape this$, float a ) {
      if ( this$ instanceof Square ) {
        return rotate$body( (Square) this$, a );
      } else if ( this$ instanceof Rectangle ) {
        return rotate$body( (Rectangle) this$, a );
      } else {
        return rotate$body( this$, a );
      }
    }

    private static Shape rotate$body( Shape this$, float a ) {
      /* method 1 body, substituting this$ for this */
      ...
    }

    private static Shape rotate$body( Rectangle this$, float a ) {
      /* method 2 body, substituting this$ for this */
      ...
    }

    private static Shape rotate$body( Square this$, float a ) {
      /* method 3 body, substituting this$ for this */
      ...
    }
  }
}
```

Fig. 25. Example showing (a) an external method family in MultiJava, and (b) its translation to Java

with internal method families, dispatching is performed using cascaded `instanceof` tests. Since the methods do not appear in the same class as their logical receivers, the receiver argument of the call is passed as an extra argument.

To invoke an external method family, the compiled code for a client loads the dispatcher object from the anchor class's `function` field and invokes its `apply` method. So the MultiJava code:

```
Shape shape1 = new Rectangle();
shape1.rotate(90.0);
```

is translated to:

```
Shape shape1 = new Rectangle();
rotate$anchor.function.apply(shape1, 90.0);
```

By typechecking requirement R3, all external methods will be declared in the same compilation unit as their top method. However, as described earlier, MultiJava allows a class declaration to contain regular Java-style methods that belong to external method families. This idiom allows a new subclass to be given appropriate overriding methods for any existing external method families. To compile these methods, MultiJava uses the Chain of Responsibility pattern [Gamma et al. 1995, pp. 223–232].

For example, Fig. 26a shows a class, `Oval`, containing a method that belongs to the external method family `rotate`. Fig. 26b shows the results of compiling `Oval`. A new nested dispatcher class, `Oval.dispatcher`, is created that implements the same interface as does the original dispatcher in `rotate`'s anchor class. The new dispatcher's `apply` method checks whether the runtime arguments should dispatch to the local `rotate` method. If there were other `rotate` methods declared in `Oval`, they would be compiled into this `apply` method as well. If no local method is applicable, the `apply` method of the dispatcher's `oldFunction` field is invoked.

For this compilation strategy to work properly, the new dispatcher's `oldFunction` field must point to the original dispatcher object, and the `function` field of `rotate`'s anchor class must be mutated to point to the new dispatcher object. Both of these tasks are accomplished during static initialization of `Oval`, as shown in the figure. These links from the `function` field to the new dispatcher object and from the `oldFunction` field to the original dispatcher object form the chain of responsibility. When the method family is invoked, the `apply` method of the new dispatcher object is called. It checks if any of its methods are applicable. If none are, it calls the `apply` method of the original dispatcher object.

This chain-of-responsibility compilation strategy works when a single subclass adds methods to the external method family, or when several subclasses do so. Each dispatcher object checks for the applicability of its methods and, if no applicable methods are found, passes control on to the next dispatcher in the chain. Eventually dispatching either finds a dispatcher object with an applicable method, or the search ends at the initial dispatcher object installed when the method family was created. Typechecking requirement R2 ensures that this last dispatcher object on the chain includes a default method that handles all arguments, guaranteeing that dispatching terminates successfully.

Java ensures that superclasses are initialized before subclasses [Gosling et al. 2000, §12.4], so dispatcher objects for superclasses will always be put onto the chain earlier than subclass dispatchers. Therefore, subclass dispatchers will be invoked before superclass dispatchers, as desired. Two unrelated classes might

(a)
```
/* Implements Oval in MultiJava */
public class Oval extends Shape {
  ...
  public Shape rotate(float a) { /* method 1 body */  ... }
}
```

(b)
```
/* Translation of Oval to Java */
public class Oval extends Shape {
  ...
  /* static initializer */
  static {
    synchronized (rotate$anchor.class) {
      rotate$anchor.function = new dispatcher( rotate$anchor.function );
    }
  }
  private static class dispatcher implements rotate$anchor.signature {
    public rotate$anchor.signature oldFunction;
    public dispatcher( rotate$anchor.signature oldF ) {
      oldFunction = oldF;
    }
    public Shape apply( Shape this$, float a ) {
      if (this$ instanceof Oval) {
        return rotate$body( (Oval) this$, a );
      } else {
        return oldFunction.apply( this$, a );
      }
    }
    private static Shape rotate$body( Oval this$, float a ) {
      /* method 1 body, substituting this$ for this */
      ...
    }
  }
}
```

Fig. 26. Example showing (a) a class extending an external method family, and (b) its translation to Java

have their dispatchers put onto the chain in either order, but this is fine because the dispatching semantics ensures that the methods of such unrelated classes are applicable to disjoint sets of valid type tuples, so at most one class's methods could apply to a given invocation.

There is a degenerate case where a superclass static initializer instantiates a subclass and invokes an external method family on the subclass instance. In our running example, this would correspond to the static initializer for `Shape` including the code `new Oval(...).rotate(90.0)`. In this case, with the compilation strategy shown thus far, it would be possible to invoke the external method family before `Oval`'s static initializer had run. We must ensure that the chain of responsibility is updated before any instances of `Oval` are available to be dispatched on. The actual implementation updates the chain of responsibility at the beginning of each of `Oval`'s constructors (with appropriate guards against multiple updates). We omit these details from the figures for clarity. Also omitted for clarity are lock-

ing mechanisms to ensure thread safety for updates and accesses of the chain of responsibility. For example, all accesses to a function field are synchronized on the `Class` object of the anchor class.

4.3 Compiling Super and Resend

The compilation of super calls and resends presents interesting challenges. Because of the various compilation tactics for method definitions, the compiled super call or resend may originate in:

—a nested class of an anchor class (for external methods of an external method family),
—a nested class of a regular Java class (for Java-style methods of an external method family), or
—in a regular Java class (for Java-style methods of an internal method family).

The target method of the invocation may appear in the same variety of locations. Thus there are a number of permutations of caller and target method locations. When the target method belongs to an internal method family we use the Java virtual machine's `invokespecial` bytecode, just as is done to compile super calls in Java [Lindholm and Yellin 2000, §6]. When the target method belongs to an external method family we use the functions stored in the chain of responsibility. In some cases the generated code must use synthetic methods that direct execution to the appropriate nested dispatcher class or must skip functions in the chain of responsibility. These corner cases are quite mundane; the details are available in Clifton's thesis [Clifton 2001, §3.4].

4.4 Other Compilation Issues

We conclude our discussion of the compilation strategy for MultiJava by addressing three additional interesting issues.

4.4.1 *Pleomorphic Methods.* In Java, it is possible for a method to simultaneously belong to more than one method family. For example, as shown in Fig. 27, a concrete method may both override a superclass method and implement an interface method. We say that such a method is *pleomorphic*.[11] Because of Java's single inheritance, only one of the method families to which a pleomorphic method belongs will have a concrete top method. The other method families must be declared in interfaces.

It is possible that a particular client only sees one of the method families containing a pleomorphic method. For example, in the code:

```
MetallicAppearance ma;
...
ma.brush();
```

the client can only see the method family declared in the interface `MetallicAppearance`. In Java, an `invokeinterface` instruction is used if the client-visible

[11]The term "pleomorphic" comes from crystalography, where it means "having more than one lattice structure."

```
public class Painter {
  public void brush() { ... }
}

public interface MetallicAppearance {
  void brush();
}

public class MetallicPainter extends Painter implements MetallicAppearance {
  /* Overides Painter.brush() and implements MetallicAppearance.brush() */
  public void brush() { ... }
}
```

Fig. 27. A pleomorphic method, belonging to more than one method family

method family is declared in an interface; otherwise, an `invokevirtual` instruction is used. Except for the instruction used, the calling convention is the same for either sort of method family, and either sort of call can resolve to the same pleomorphic method at run time [Lindholm and Yellin 2000, §6].

In MultiJava, pleomorphic methods are slightly more complicated. MultiJava's type system, like Java's, ensures that only one of the method families to which a pleomorphic method belongs will have a concrete top method; the other method families must be declared in interfaces. However, in MultiJava the concrete top method may be an external method. As we have discussed, external method families use a different calling convention than do internal method families. For a client that only sees the external method family, the pleomorphic method must appear in the external method family's chain of responsibility. But for a client that only sees the method family declared in an interface, the pleomorphic method must be defined inside the receiver class.

Our solution to this dilemma is to compile the pleomorphic method according to the strategy for external method families. Additionally, we create a *redirector method* within the receiver class that directs an invocation of the internal method family on that class into the external method family. Clifton's thesis describes this technique in more detail [Clifton 2001, §3.3].

4.4.2 *Private External Methods.* A second issue that arose during the implementation of `mjc` was the handling of private external methods. MultiJava allows a compilation unit declaring a public external method families to also declare private helper methods. A sample of a compilation unit with a private external method is given in Fig. 28. Our compilation strategy is to make the anchor class of the private external method be a nested class of the regular anchor class. For the code in the figure a `swap$anchor` nested class is created inside the `sort$anchor` class. This enforces the access semantics for the private external method and avoids a name clash should a non-private external method family named `swap` be declared in the same package.

4.4.3 *Multimethod and External Method Signatures in Bytecode.* A third issue that arose during our implementation was the need to read MultiJava-specific features from bytecode. This is necessary to allow compilation units to be safely compiled given only the bytecode, but not the source, of other compilation units

```
/* In file "sort.java" */
public void List.sort() {
  ...
    this.swap(i,j);
  ...
}

/* private external helper method */
private void List.swap(int i, int j) {
  Object temp = get(i);
  set(i, get(j));
  set(j, temp);
}
```

Fig. 28. Compilation unit with a private external method

that are depended upon. Our solution to this problem is to use the capability of adding custom attributes to bytecode [Lindholm and Yellin 2000, §4.7.1]. We use attributes to encode the signatures of all the local methods of an external method family in the bytecode for the method family's anchor class. Similarly, we use attributes to encode the signatures of all multimethods (that are not also external methods) in their receiver class's bytecode. These attribute values are easily read from bytecode, allowing a MultiJava compiler to efficiently retrieve this information. Using these attributes does not cause any incompatibilities with existing virtual machine implementations; such implementations must ignore attributes that are not recognized.

5. APPLICATIONS

MultiJava has been used by others in the implementation of several applications. This section illustrates the ways in which MultiJava's features have been employed and reports on user feedback about the benefits and limitations of the language. The applications span several domains. First, MultiJava has been used to implement reliable ubiquitous computing systems. Versions of the following systems have been implemented at least in part using MultiJava:

—Guide [Philipose et al. 2004] is a system for inferring the presence and nature of human activities in indoor spaces.

—Labscape [Arnstein et al. 2002] is a ubiquitous computing environment for cell biology laboratories.

—The Location Stack [Hightower et al. 2002] is a framework for combining and representing measurements from a heterogeneous network of sensors that track the locations of objects in an environment.

Second, MultiJava has been used to implement two compilers. Hydro is a domain-specific language for XML data processing, and HydroJ [Lee et al. 2003] is an extension to Java supporting Hydro's features.

Finally, MultiJava was used to implement a graphical user interface (GUI) for manipulating a reconfigurable chip that performs machine learning tasks [Bridges et al. 2003].

```
class Measurement {
  boolean equals(Object o) {
    if (o instanceof Measurement) {
      Measurement m = (Measurement) o;
      /* check equality of two Measurements */
      ...
    } else {
      return false;
    }
  }
}
```

(a)

```
class Measurement {
  boolean equals(Object@Measurement m) {
    /* check equality of two Measurements */
    ...
  }
}
```

(b)

Fig. 29. (a) Binary methods in Java, and (b) in MultiJava

5.1 Multimethods

Our users employ multimethods in several ways. As expected, multimethods are used to solve the binary method problem. Multimethods have also proven useful in implementing event handlers, performing tree traversals, and in finite-state machines. The following sections detail these uses.

5.1.1 *Binary Methods.* One of the simplest uses of multimethods is in the implementation of binary methods. In Java, all classes have at least one binary operation, the `equals` method family inherited from `java.lang.Object`. Fig. 29a shows a common idiom for implementing `equals` methods in Java, and Fig. 29b shows the MultiJava equivalent. `Measurement` is the base class for sensor measurements in the Location Stack. Multimethods are used in this way to implement `equals` in this and many other classes of the Location Stack, as well as in the implementations of the Hydro and HydroJ compilers. Hydro and HydroJ also include an expressive sublanguage for pattern matching on XML data. A variant of the MultiJava style shown in Fig. 29b is used in the Hydro and HydroJ compilers to implement binary methods on patterns, including pattern specificity checking and pattern intersection.

Even on the small example in the figure, the MultiJava version enjoys several advantages. The method in the MultiJava version declaratively expresses its dispatching behavior in its header. This allows programmers to more easily understand the functionality and allows MultiJava to check statically for incompleteness and ambiguities. The MultiJava version also allows functionality to be more easily inherited. For example, the `else` case in Fig. 29a is not needed in the MultiJava version. Instead, MultiJava's dispatching semantics ensures that the `equals` method in `java.lang.Object` will be invoked whenever the receiver is a `Measurement` instance (or a subclass) but the argument is not. Because `Object`'s `equals` method implements pointer equality, the simplified code has the same semantics as the original. In contrast, the Java version of `Measurement`'s `equals` method must always include an `else` case to ensure completeness.

5.1.2 *Event Dispatching.* Many kinds of applications are naturally structured in an event-based style. In this style, components do not communicate by directly sending messages to one another. Instead, each component is able to *announce*

```
abstract class Component {
  abstract void handleEvent(Event e);
}
```

Fig. 30. Base class of components in an event-based system

a set of events. Separately, other components can register to receive notification whenever a certain event is announced by providing a *handler* procedure for the event. When an event is announced, the system invokes all the handlers that are associated with that event. The canonical example of an event-based system is a GUI. Events are announced in response to user actions (e.g., clicking a button), and these events trigger the appropriate actions of components (e.g., updating the display). Java's Abstract Windowing Toolkit (AWT) is a library for building GUIs that employs the event-based style.

Several of the projects in our user community employ event-based architectures. The GUI for the reconfigurable chip is built on top of AWT. The event-based style is also used by all of the ubiquitous systems described earlier. Each of those systems is built on top of either `one.world` [Grimm et al. 2001] or Rain [Rain 2004], which are event-based libraries for facilitating the creation of ubiquitous computing applications in Java. In the ubiquitous systems, extensibility is at a premium: it must be possible for new components to easily join and leave the system dynamically. The event-based style facilitates this extensibility by keeping components loosely coupled, since components communicate only indirectly through events.

5.1.2.1 *Basic Event Dispatching.* In the context of an object-oriented language like Java, an event-based system typically includes an abstract class or interface that defines the required functionality of all components, as shown in Fig. 30. Each component is a concrete subclass of `Component`, and each event is similarly a subclass of an abstract `Event` class. A component's `handleEvent` method is its event handler: when an event occurs, the system notifies those components that have registered for the event by calling their `handleEvent` methods. The object representing the event is passed as an argument. This structure is used by both the `one.world` and Rain libraries.

The `handleEvent` operation is a natural application for multiple dispatch in MultiJava. The functionality for handling an event depends both on which component is handling the event and on which event has been announced, but Java only allows one of these hierarchies to be dispatched upon. Therefore, programmers must manually dispatch on the other hierarchy, usually via runtime type tests and casts. An example of Java and MultiJava event handlers in a hypothetical GUI for a text editor is shown in Fig. 31.

The benefits of MultiJava illustrated for binary methods are accrued to an even greater extent for event handling. MultiJava allows each conceptual handler to be encapsulated as its own method, rather than buried in an `if` case of one monolithic method. Users report that this style of implementing handlers exactly matches their high-level view of a component as containing a set of handlers, each handling a particular event. The header of each multimethod characterizes the conditions under which that handler will be invoked, and static checking ensures that there

```
class EditorGUI extends Component {
  void handleEvent(Event e) {
    if  (e instanceof Open) {
      Open o = (Open) e;
      /* open a file */
      ...
    } else if (e instanceof Save) {
      Save s = (Save) e;
      /* save the currently opened file */
      ...
    } else if (e instanceof Quit) {
      Quit q = (Quit) e;
      /* quit the application */
      ...
    } else {
      /* handle unexpected events */
      ...
    }
  }
}
```

(a)

```
class EditorGUI extends Component {
  void handleEvent(Event@Open o) {
    /* open a file */
    ...
  }
  void handleEvent(Event@Save s) {
    /* save the currently opened file */
    ...
  }
  void handleEvent(Event@Quit q) {
    /* quit the application */
    ...
  }
  void handleEvent(Event e) {
    /* handle unexpected events */
    ...
  }
}
```

(b)

Fig. 31. (a) Event handling in Java, and (b) in MultiJava

is a most-specific applicable handler for each possible event. As a simple example of static checking, the lack of a default handler accepting any `Event` would signal a static completeness error in MultiJava. In contrast, the Java version would compile without error but fail dynamically in unexpected ways if the final `else` case were missing. Users report such errors to be common, particularly in the ubiquitous computing context, where the system can easily become misconfigured as components enter and exit.

The MultiJava style also had the unexpected effect of encouraging programmers to write better documentation. For example, "Handles events" is the typical comment for a monolithic `handleEvent` method. A user reported that MultiJava made it natural for him to instead document the actual behavior of each handler in comments. Such documentation can then be displayed by `mjdoc`, an HTML-based documentation tool for MultiJava programs developed by David Cok, which is similar to Java's `javadoc` tool. (The `mjdoc` tool was developed based on user requests for such a facility.)

5.1.2.2 *Event Dispatching in Practice.* Event handlers are often significantly more complicated than the example shown in Fig. 31, and MultiJava's advantages over Java increase with this complexity. For example, the various `Event` subclasses may form a deep hierarchy, with subclasses of `Event` having their own subclasses. This scenario occurs in the Location Stack. In the text editor example above, suppose that `Save` has a subclass `SaveAs` for saving to a new file. If special behavior for `SaveAs` is required for `EditorGUI` in Fig. 31, the Java version must be updated carefully with a new `if` case. The programmer must ensure that that the `SaveAs` case comes before the case for `Save`, or else the new case will never be invoked. In general, the programmer must always ensure that a class is tested before any of its superclasses. In the MultiJava version, a new `handleEvent` multimethod special-

izing on `SaveAs` can be added anywhere in `EditorGUI`, and MultiJava's symmetric dispatching semantics ensures that it will be invoked properly.

Code for deep hierarchies can also naturally take advantage of `resend`s. For example, suppose that `EditorGUI` on `SaveAs` should do everything that it does for `Save`, preceded by some extra statements $s_1, \ldots, s_n$ (e.g., initializing the new file) and followed by some other statements $s_{n+1}, \ldots, s_m$. In the MultiJava version, the `handleEvent` method for `SaveAs` simply performs $s_1, \ldots, s_n$, uses `resend` to invoke `EditorGUI`'s handler for `Save`, and then performs $s_{n+1}, \ldots, s_m$. The Location Stack employs this style to reuse code across handlers. In contrast, in the Java version one must either duplicate the code for `Save` in the case for `SaveAs`, or one must create a helper method that both cases invoke, which requires that the case for `Save` be modified.

Another common source of complexity in practice is when a handler depends upon more than just the kind of event announced in order to determine what action to take. For example, events in the `one.world` library have a `closure` field of type `Object`. The closure is used in request-response interactions to distinguish among several response events processed by the same event handler. Many handlers dispatch based on both which event is passed and what kind of closure that event contains. MultiJava generalizes naturally to handle this scenario, via dispatch on multiple arguments. Instead of defining `handleEvent` by a set of multimethods, a single `handleEvent` method now invokes a helper method `handleWithClosure`, passing both the announced event and the event's closure field. The `handleWithClosure` method then performs the desired dispatching:

```
void handleWithClosure(Event@Event1 e1,
                       Object@LocalClosure closure) { ... }
void handleWithClosure(Event@Event2 e2,
                       Object@RemoteClosure closure) { ... }
...
```

Each `handleWithClosure` method cleanly documents the conditions under which it will be invoked, and static typechecking ensures a most-specific applicable handler for each (event, closure) pair. The only disadvantage is the need to create this helper method, in order to dispatch on the `closure` field. Aside from being more verbose, the helper method also breaks the bond between the event and the closure: there is nothing in the `handleWithClosure` operation documenting the fact that the closure argument should be the value of the `closure` field of the event argument. Using the helper method also makes it hard to inherit handlers from superclasses (as described below), unless those superclasses also use `handleWithClosure` helper methods.

Despite these disadvantages, the MultiJava solution is much more elegant than an equivalent solution in Java. To handle dispatch on `closure` fields in Java, the event-handling code must be further obfuscated with additional type tests and casts. Dispatch on `closure` fields was typically implemented in Java using nested `if`s: an outer `if` dispatched on the event, and each case of that `if` included code to dispatch on the various kinds of closures. Such an approach is not just tedious but is asymmetric, making it harder to understand the conditions under which

each "handler" triggers. Further, a `one.world` user reported that this style made it too easy to omit enforcement of some dispatching requirements in the code. For example, dispatching on the closure could be omitted in outer `if` cases that didn't use the closure (or didn't depend on its runtime type), even though it was still intended that the closure be of a particular type. Because of the tedium of the Java style, such errors of omission occurred often in that user's Java code. Also, the user reported that on revisiting code that omitted dispatching on closure types, he was sometimes unsure whether he had intentionally or inadvertently omitted the closure dispatching. With MultiJava, there is no advantage to omitting dispatch requirements, because each `handleWithClosure` method explicitly mentions both the event and its closure, and expressing a dispatching requirement is lightweight and declarative.

5.1.2.3 *Component Hierarchies.* MultiJava also allows new ways of structuring handlers that were not considered previously by the developers of the event-based systems. Because multimethods can be inherited, it is possible to have deep hierarchies of components. Each component inherits all of the handlers of its superclasses, optionally overrides any of these handlers, and adds new handlers.

Both the reconfigurable chip's GUI and the Location Stack employ this style. For example, in the GUI, the abstract `Component` class represents an arbitrary GUI element. It has a default method handling any `Event` that acts as the "error handler," freeing subclasses from having to handle unexpected events. An abstract `HighlightedComponent` subclass represents a component that is able to be highlighted. It inherits the error handler and adds a handler that responds to the act of highlighting by updating the GUI properly. A `TerminalComponent` is a subclass of `HighlightedComponent` representing a wire connection point on the chip. It inherits the error handler and the highlighting functionality, and it has additional handlers for events specific to terminals.

Simulating this idiom of fine-grained handlers inheriting functionality from superclasses is very awkward in the Java version, where each handler is a monolithic `if` block. Each subclass has to explicitly invoke `super` in the right places to manually dispatch to superclass handlers when inheritance is desired. That style is so unnatural that the developers of these systems did not even consider handler inheritance to be an option before they started using MultiJava.

5.1.2.4 *Value Dispatching.* The event-based style presented so far uses a hierarchy of `Event` subclasses to represent the various events in a system. An alternative approach employs a single `Event` class with no subclasses, with an integer or string field signifying which kind of event a particular instance represents. Although this latter style is less object-oriented and less expressive (e.g., it does not allow deep hierarchies of events), it is perceived to be a more lightweight solution and is fairly common. Event-based systems that employ this style can naturally use MultiJava's value dispatching to declaratively dispatch on the event "tags."

For example, the Java AWT sometimes uses strings to distinguish events. Its `ActionEvent` class has an "action command" string that is set in the constructor and specifies the event being represented. Fig. 32 shows how event handlers in this style are written in Java and MultiJava, for the hypothetical text editor. The

```
void handleEvent(ActionEvent e) {
  String cmd = e.getActionCommand();
  if  (cmd.equals("open")) {
    /* open a file */
    ...
  } else if (cmd.equals("save")) {
    /* save the currently opened file */
    ...
  } else if (cmd.equals("quit")) {
    /* quit the application */
    ...
  } else {
    /* handle unexpected events */
    ...
  }
}
```

(a)

```
void handleEvent(ActionEvent e) {
  handleCmd(e, e.getActionCommand());
}
void handleCmd(ActionEvent e,
     String@@"open" cmd) {
  /* open a file */
  ...
}
void handleCmd(ActionEvent e,
     String@@"save" cmd) {
  /* save the currently opened file */
  ...
}
void handleCmd(ActionEvent e,
     String@@"quit" cmd) {
  /* quit the application */
  ...
}
void handleCmd(ActionEvent e,
     String cmd) {
  /* handle unexpected events */
  ...
}
```

(b)

Fig. 32. (a) Dispatching on primitive events in Java, and (b) in MultiJava

MultiJava version has the same advantages as described for the code in Fig. 31b. Labscape employs MultiJava in this way to handle events related to its GUI. Value dispatching allows Labscape to use the existing AWT library while still enjoying the benefits of the MultiJava style.

There are opportunities for value dispatching even when events are written in a class hierarchy. An earlier example illustrated event handlers that depend upon an event's `closure` field in addition to the event's runtime type. Some `one.world` events, subclasses of `TypedEvent`, contain an integer `type` field which is used to distinguish among events of the same general kind. MultiJava handlers in this case look similar to the `handleWithClosure` methods presented above, but with the second argument employing value dispatching on the `type` field of the received event.

5.1.2.5 *Limitations.* Users did point out some limitations of MultiJava in the context of event dispatching. First, MultiJava can be more verbose than equivalent code using `if`s, because of the need to repeat the method's header for each handler. Second, it is easy to forget the `Event@` portion of a formal parameter's type, thereby accidentally using static overloading instead of multimethod dispatch. To alleviate this problem, we augmented the `mjc` compiler to signal a warning if static overloading is used where multimethod dispatch could be used instead. Third, dispatching on properties of an event other than its runtime type requires the creation of helper methods, as mentioned earlier.

Fourth, it is not easy to update a component when a new event type enters the system. The component must be augmented in place to contain a multimethod

specializing on the new event type. This is still better than the Java version, in which the programmer must find the right place in the `if` chain to place a new case. However, it would be nice to write the new multimethod external to the component, thereby allowing new events to be incorporated without modifying existing code. This ability is a step toward allowing a running system to be updated on the fly with new events, which is important for the ubiquitous computing applications.

Finally, the need for default methods limits MultiJava's ability to perform useful completeness checking. It is impossible for a component to document the fact that it handles exactly three kinds of events, and no others. Instead, it must always include a default method, to handle any unexpected events. Some programmers found it useful for the language to force them to think about exceptional situations, but others thought it more of a nuisance. An extension to MultiJava described in Section 6 below, called Relaxed MultiJava, provides one possible solution to this problem.

5.1.3 *Noninvasive Visitors.* Section 2.3 illustrated how multimethods can be used to implement a noninvasive version of the visitor design pattern. The HydroJ compiler is built on top of the Polyglot extensible compiler framework [Nystrom et al. 2003], which supports visitors over the hierarchy representing abstract syntax tree (AST) nodes. HydroJ provides new subclasses of Polyglot's visitors and uses multimethods to implement the node dispatch.

As described in Section 2.3, open classes enjoy two key advantages over noninvasive visitors. However, the noninvasive visitor pattern provides the benefits of classes, which external methods lack. Helper fields can easily be included in a visitor class, while these must be simulated through extra parameters in external methods.[12] More importantly, visitors can inherit functionality from superclasses, analogous to the inheritance of event handlers described earlier. For example, Polyglot provides an abstract `HaltingVisitor` class, which performs a boilerplate traversal over AST nodes that also supports bypassing certain nodes during the traversal. Concrete visitors that require the functionality to bypass nodes simply subclass `HaltingVisitor` and provide overriding methods to customize its behavior as necessary. In contrast, each external method family must implement its own traversal behavior from scratch.

One limitation of MultiJava that arises in the HydroJ compiler is the requirement that specializers be classes. Polyglot provides its AST nodes as a hierarchy of interfaces, with the intent that the associated implementation classes should remain hidden from clients. The HydroJ compiler must break this abstraction boundary and dispatch directly on the implementation classes. The Java style does not suffer from this problem, because `instanceof` tests are allowed on interfaces. Again, Relaxed MultiJava provides one solution to this problem (see Section 6).

5.1.4 *Finite-state Machines.* A common way to implement a finite-state machine (FSM) in Java is to associate an integer constant with each state. The FSM's

[12] Analogous to external methods, external fields would also fit this purpose. The key obstacles in implementing external fields are initializing those fields and handling persistence for them. These issues are discussed in more detail in Clifton's thesis [2001, §6.1.4]. A thorough investigation of these issues remains as future work.

```
class ZeroOneFSM {
  static final int EXPECT_ZERO = 0;
  static final int EXPECT_ONE = 1;

  int currState = EXPECT_ZERO;
  int numOccurrences = 0;

  void readAndTransition(int input) {
    transition(input, currState);
  }

  void transition(int@@0 input, int@@EXPECT_ZERO state) {
    currState = EXPECT_ONE;
  }
  void transition(int@@1 input, int@@EXPECT_ONE state) {
    currState = EXPECT_ZERO;
    numOccurrences++;
  }
  void transition(int input, int state) {
    currState = EXPECT_ZERO;
    numOccurrences = 0;
  }
}
```

Fig. 33. Implementing finite-state machines in MultiJava

class has a field recording the current state, and a method in the class implements the FSM's transition function: based on the given input and the current state, the method performs some actions and transitions to a new state. This style is error prone and difficult to understand, since the programmer must manually implement the logic of the transition function as a monolithic block of code. An alternative approach uses the "state" design pattern [Gamma et al. 1995]. This pattern uses an explicit class hierarchy to represent states, and each state class implements its portion of the FSM's transition function. However, the state pattern is heavyweight and tedious, requiring the introduction of several new classes and requiring the FSM to explicitly forward messages to its state field. Further, while the state pattern makes dispatch on the FSM's state declarative, dispatch on the FSM's input must still be manually implemented.[13]

In MultiJava, value dispatching provides a natural way to implement FSMs that are as lightweight as the first style described above and as declarative as the second style. As a simple example, Fig. 33 implements an FSM that keeps track of the number of consecutive alternations of 0 and 1 that have been input. There are two states, which respectively track whether a 0 or 1 is expected as input. The transition function has three transitions, each nicely encapsulated in its own multimethod. For example, the first `transition` method "fires" when the input is 0 and the FSM is in the EXPECT_ZERO state. In that case, the FSM moves to the EXPECT_ONE state. The method uses MultiJava's allowance of any compile-time constant expression after a `@@`. The last `transition` method is the one required

[13]The standard version of the state design pattern uses a separate method for each input, rather than a single transition function. Using separate methods works well if there is a fixed set of inputs but makes it difficult to extend this set after the fact.

```
interface Procedure {
  void apply(Object o);
}

void Iterator.doEach(Procedure p) {
  while (this.hasNext()) {
    p.apply(this.next());
  }
}

/* sample client code */
Iterator i = ...
i.doEach(new Procedure() {
  void apply(Object o) { ... } });
```

Fig. 34. Adding closure-based iteration to Java

by typechecking requirement R1, to ensure completeness. It handles the case when unexpected data is input (or an unexpected state is reached), in which case the FSM resets.

The Location Stack uses this style to implement the FSMs that parse readings from the various location sensors. The developer reports that it is much easier to understand the behavior of an FSM written in this way, versus the typical Java style. The MultiJava version also enjoys all the benefits described earlier for multimethods. For example, the FSM is easily extensible by subclasses, which can add new transitions and optionally override existing ones.

5.2 Open Classes

The Hydro and HydroJ compilers exploit open classes for several purposes, which are described in this section.

5.2.1 *Unavailable Source.* A common use of open classes has been to augment classes whose source is not available (or not easily modified). The Hydro and HydroJ compilers add several methods to classes in the Java standard library. For example, a method for removing whitespace from a string is defined in HydroJ as follows:

```
public String String.deleteWhitespace() { ... }
```

Clients can import the new method family and then invoke it as if it were part of the original functionality of strings. In Java, this idiom is typically simulated by a static method in a dummy class, which is a bit more tedious and has a different call syntax from the original methods of `String`.

A more interesting example is illustrated in Fig. 34, which is a variant of code from the Hydro compiler. The `doEach` method augments `java.util.Iterator` with closure-based iteration. The closure is defined as a (typically anonymous) class that implements the `Procedure` interface. A side benefit of open classes illustrated in this example is the ability to add methods to interfaces (like `Iterator`). Such an ability was absent in the original design of MultiJava, but was added based on user feedback.

```
/* file "print.java" */
public String TreePatternNode.print() { ... }
public String ListPatternNode.print() { ... }
public String StarPatternNode.print() { ... }
...
```

Fig. 35. Structuring code by algorithm with open classes

The Hydro compiler offers yet another interesting example related to unavailable source. Hydro uses the SableCC parser generator [Gagnon and Hendren 1998], which builds the AST node hierarchy automatically from a description of Hydro's grammar. Modifying the resulting AST classes is undesirable, because any changes will be lost the next time SableCC is run. The nodes generated by SableCC provide a visitor-like framework so clients can implement external traversals over the AST hierarchy, and the Hydro developer used these visitors to implement the major passes in the compiler. However, he preferred using open classes for functionality that is not meant to traverse the entire AST hierarchy. For example, Hydro includes a rich language for pattern matching, and open classes make it easy to add new behavior to the pattern nodes. Using the visitor infrastructure would require that the external operations for patterns actually be able to handle an arbitrary node, which is more tedious and loses some static type safety.

A necessary limitation of open classes is the lack of privileged access to the receiver class. To use open classes successfully, the public functionality of the receiver must be rich enough to allow clients to implement unanticipated behaviors. For example, `String` has methods that provide access to each character, and this is enough to allow whitespace to be removed by clients, as shown in the first example above. This limitation of open classes is necessary to retain Java's encapsulation properties, and it is shared by the Java solutions to the augmenting method problem discussed in Section 2.2.1.

5.2.2 *Client-specific Extensions.* It can make sense to make an operation external even if the source code for its receiver class is available. One such scenario is when the new functionality is client-specific rather than general-purpose. With open classes, the new functionality can be implemented without polluting the view of the original receiver class as seen by other clients. In general, each client can have his own library of extensions to an existing class hierarchy. The HydroJ compiler implements client-specific operations in this way. For example, the compiler maintains an instance of `java.util.List` containing AST nodes. The compiler extends Java's `List` implementation with an external method for deep copying `List`'s of AST nodes. Although in this case the source code for `List` is actually not available for editing, the developer reports that he would use external methods for this operation even if he had source-code access to `List`. Indeed, an AST-specific operation does not belong in the view of `List` as seen by all clients.

5.2.3 *Flexible Code Structuring.* Open classes also allow code decompositions other than by receiver class. It is sometimes useful to encapsulate an entire method family's methods as a unit, rather than spreading the methods across the various receiver classes. This can be especially helpful when the methods implement a

single conceptual algorithm. As an example, the code for printing AST nodes in the HydroJ compiler is implemented as an external method family, a portion of which is illustrated in Fig. 35. The developer felt that this decomposition was more natural than the by-class view. In addition, the HydroJ compiler contains a few different algorithms for printing AST nodes. Each is implemented as an external method family, and clients import the one appropriate to their needs.

6. RELAXED MULTIJAVA

As discussed above, the experience of our user community provides a practical demonstration of several ways in which MultiJava can be used to improve code comprehension, extensibility, and correctness. User experience with MultiJava has also helped to identify useful enhancements to the language. In the previous section, we mentioned two of these: David Cok's HTML documentation utility, `mjdoc`, and the addition of external methods on interfaces. In this section we discuss an extension to MultiJava, called Relaxed MultiJava (RMJ) [Millstein et al. 2003], that relaxes the static typechecking requirements in order to address other concerns identified by our users.

The key observation behind RMJ is that violations of MultiJava's typechecking requirements, discussed in Section 3.3.2, indicate only the *potential* for a method family to be incompletely or ambiguously implemented, because MultiJava must be conservative given only a modular view of the program. Therefore, RMJ performs the same modular static typechecking as MultiJava, but RMJ treats a violation of some requirement as a warning rather than an error. The programmer can choose to resolve the violation, as he would be forced to do in MultiJava, thereby obtaining a modular guarantee of type safety. Alternatively, the programmer can choose to violate the MultiJava requirement and take responsibility for ensuring that the potential error does not arise, in order to obtain the desired expressiveness. A custom class loader [Liang and Bracha 1998] for RMJ incrementally checks that potentially erroneous method families remain complete and unambiguous as classes are loaded, thereby ensuring that all errors are still detected no later than class load time. Therefore, as in MultiJava, message-not-understood and message-ambiguous errors cannot occur when messages are sent at run time.

RMJ provides several useful programming idioms that are disallowed in MultiJava. Treating requirement R2 as a warning instead of an error allows programmers to implement abstract external methods. For example, when the compilation unit in Fig. 21 is typechecked, RMJ signals a warning but still allows compilation to complete. The abstract `circumference` method documents the fact that each concrete `Shape` subclass must provide its own `circumference` method (or must inherit one). The RMJ class loader ensures this is the case, incrementally checking concrete `Shape` subclasses as they are loaded in the program. In this way, RMJ relieves the burden on the programmer of having to write default methods when they are deemed unnecessary or unnatural. In our example, unless the signature of the original `Shape` class is very rich, it is unlikely that a reasonable default implementation of `circumference` can be provided. In MultiJava, the programmer may therefore have no choice but to make the default method raise an exception, which is effectively the same as the message-not-understood error that MultiJava's restrictions

```
package client;

import examples.circumference;
import Circle;

public double Circle.circumference() {
  return 2.0 * Math.PI * radius();
}
```

Fig. 36. Glue methods in RMJ

are meant to prevent. RMJ avoids such awkward default methods, at the cost of some additional load-time checking.

As another example, consider the `Circle` class in Fig. 8. The implementers of the `Circle` and `circumference` extensions to the original `Shape` class are unaware of one another. Therefore, neither extension provides a method for computing the circumference of a `Circle`. Given the `circumference` methods in Fig. 21, RMJ will signal an incompleteness error if `circumference` and `Circle` are ever loaded in the same program. However, in RMJ a client who wants to include both extensions in a program can make them work together by implementing the appropriate method, as shown in Fig. 36. MultiJava would disallow this method from being written, since it violates requirement R3 — the external method is not declared in the same compilation unit as its method family's top method. In RMJ, the method triggers a compile-time warning but is allowed, and the RMJ class loader incrementally checks that the method is not ambiguous with any modularly unseen `circumference` methods. We call methods that violate requirement R3 *glue methods*, because they serve to integrate two previously independent libraries.

RMJ's typechecking relaxations are implemented as an option to the `mjc` compiler. We have also implemented `RMJClassLoader`, a subclass of Java's default class loader, in order to perform RMJ's incremental load-time checks. While the custom class loader naturally augments Java's dynamic loading scheme, users may sometimes desire early feedback about the possibility of load-time errors. We therefore also provide a "preloader" tool `RMJPreLoader`, which uses whole-program information to statically check for the possibility of such errors. If no errors are found, then the programmer is assured that the `RMJClassLoader` will never signal an error, for any possible run of the given program (modulo the use of reflection to dynamically load classes).

Although RMJ offers strictly greater flexibility than MultiJava, we have chosen not to make the relaxed version of the language the default. To implement glue methods in their full generality, the current code generation strategy for RMJ produces separate dispatcher classes for each method of an external method family. This strategy results in bytecode that is slower than that currently produced by the non-relaxed option, which compiles all external methods from the same compilation unit into a single dispatcher class. While MultiJava could be made to use RMJ's compilation strategy, we have chosen not to impose the efficiency penalty on the common case of complete, unambiguous method family implementations.

The introduction of glue methods also requires augmented support for loading and linking external methods into applications. Currently RMJ's custom class

Table I. Comparison of dispatch times for a simple tree walk

Implementation	5 nodes	7 nodes	341 nodes
Extensible Visitor	50 ms	80 ms	3,265 ms
Open Classes	270 ms	311 ms	15,542 ms
Speed Up	0.19	0.26	0.21

loader is responsible for explicitly loading all external methods when appropriate, in addition to performing load-time checks on classes. Therefore, in order for external methods to work properly, all RMJ programs must be run on top of our custom class loader. Keeping MultiJava, rather than RMJ, as the default version of the language allows external methods to be loaded as described in Section 4, avoiding the need to use the RMJ custom class loader in the common case. If Java had a whole-program link-time phase in place of its lazy class-loading scheme, this distinction between MultiJava and RMJ would not exist.

7. PERFORMANCE

As stated in the introduction, the performance of MultiJava code is not a primary focus of our research. This is partly because:

—bytecode for regular Java code is no different when compiled in MultiJava, and

—MultiJava source code using multiple dispatch or open classes cannot easily be expressed in regular Java.

Nevertheless, Clifton's thesis examined the performance of MultiJava code on a variety of tasks and compared it to the performance of various partial solutions written in Java [Clifton 2001, pp. 83–88]. We include the main results here and refer the reader to his thesis for a description of the experimental technique and rationale.

Because the MultiJava compilation strategy uses typecases for multimethods, the performance of multimethods is the same as for typecases. That is, there is no additional cost for multimethods beyond what would be incurred for a Java implementation that required dispatch on non-receiver argument types.

To evaluate the performance of open classes, Clifton compares the performance of various operations implemented in MultiJava and using the Extensible Visitor Pattern [Krishnamurthi et al. 1998]. Extensible Visitor was chosen because it provides similar extensibility as open classes (though it requires advance planning and is tedious to implement). The results in Table I and Table II demonstrate that as the complexity of the operation to be performed increases, the relative disadvantage of the dispatch strategy for external method families decreases. That is, dispatch becomes a smaller percentage of running time. Table III demonstrates an interesting result. For the operation studied here, because of the additional code required for the visitor implementation (for the construction of additional visitors, the passing of arguments, and returning of results), MultiJava's open class solution is actually more efficient. In fairness, the extensible visitor code used in this last comparison was written rather mechanically and could be made more efficient with a little effort. On the other hand, the code used is representative of what one might actually write.

Table II. Comparison of dispatch times for tree size calculation

Implementation	5 nodes	7 nodes	341 nodes
Extensible Visitor	120 ms	160 ms	7,461 ms
Open Classes	251 ms	310 ms	15,783 ms
Speed Up	0.48	0.52	0.47

Table III. Comparison of dispatch times for pretty print operation

Implementation	5 nodes	7 nodes	341 nodes
Extensible Visitor	1,792 ms	2,254 ms	141,804 ms
Open Classes	1,322 ms	1,763 ms	129,135 ms
Speed Up	1.36	1.28	1.10

Table IV. Comparison of augmenting method dispatch times for varying degrees of modularity

Implementation	Time	Speed Up
Regular Methods	1,061 ms	–
Visitor Pattern	1,703 ms	0.62
Extensible Visitor	1,792 ms	0.59
Open Classes	1,322 ms	0.80

Clifton also compares MultiJava solutions to the augmenting and binary method problems with Java-based solutions that are less modular. Results for these experiments are given in Table IV and Table V.

8. RELATED WORK

This section discusses other multimethod-based languages, other approaches to the extensibility problem, and languages for advanced separation of concerns.

8.1 Other Multimethod-based Languages

There are several other languages supporting multimethod dispatch. Cecil [Chambers 1992; 1997] is a statically typed, prototype-based object-oriented language supporting multimethods written external to their associated objects. Cecil requires the whole program to safely perform implementation-side typechecking [Litvinov 1998]. Dubious [Millstein and Chambers 2002; Millstein 2003] was designed as a distillation of Cecil to its core constructs, for formal study of the modular typechecking problem. MultiJava's modular type system is based on that of Dubious.

Common Lisp [Steele Jr. 1990; Paepcke 1993] and Dylan [Shalit 1997; Feinberg et al. 1997] are both multimethod-based languages. All methods are written external to their classes. To avoid runtime ambiguities, Common Lisp totally orders the arguments of a method family; Dylan uses the symmetric semantics, as in MultiJava. Both Common Lisp and Dylan totally order the inheritance hierarchy, eliminating the potential for multiple-inheritance ambiguities. The languages

Table V. Comparison of binary method dispatch times for varying degrees of modularity

Implementation	Time
Double-dispatching	2,704 ms
Multiple Dispatch	4,306 ms
Speed Up	0.63

are dynamically typed, so they do not consider the issue of static typechecking, modular or otherwise.

Polyglot [Agrawal et al. 1991] is a database programming language akin to Common Lisp with a first-order static type system. There are no abstract methods, so there is no possibility of message-not-understood errors. Further, the dispatching semantics uses Common Lisp-style total ordering of multimethod arguments and inheritance, avoiding all ambiguities. Therefore, only the monotonicity of the result types [Castagna et al. 1995; Reynolds 1980] of multimethods needs to be checked to ensure modular type safety.

Kea [Mugridge et al. 1991] and Tuple [Leavens and Millstein 1998] are statically typed, class-based languages with symmetric multimethods. Kea has a notion of separate compilation, but this requires runtime implementation-side typechecking of method families. Tuple requires the whole program to be available in order to perform implementation-side typechecking statically. In Tuple, all multimethods are written as external methods that dispatch on an explicit tuple of arguments as the receiver, thereby cleanly separating specialized from unspecialized argument positions. An early design for MultiJava adapted this style, but we rejected that in favor of the current design for several reasons. Tuple requires that all methods of the same method family have identical specialized and unspecialized argument positions. This means, for example, that a multimethod cannot override an existing Java method. Further, because the distinction between specialized and unspecialized arguments is visible to clients in Tuple, specializing an unspecialized argument, or the converse, requires modifying all call sites in the program.

Encapsulated multimethods [Castagna 1995; Bruce et al. 1995] are a design for adding multimethods to an existing single dispatch object-oriented language. An encapsulated multimethod is written inside of its receiver's class; external methods are not supported. Encapsulated multimethods involve two levels of dispatch. The first level is just like regular single dispatch to the class of the receiver object. The second level of dispatch is performed within this class to find the best multimethod applicable to the dynamic classes of the remaining arguments. The encapsulated style can lead to duplication of code, since multimethods in a class cannot be inherited for use by subclasses.

Several other efforts have extended Java to support multimethod dispatch. Parasitic methods [Boyland and Castagna 1997] and Half & Half [Baumgartner et al. 2002] are both extensions to Java that include encapsulated multimethods. Both augment the encapsulated style with the ability to inherit multimethods from superclasses. The resulting expressiveness is comparable to that of MultiJava's internal multimethods, but MultiJava additionally retains the natural symmetric multimethod dispatch semantics. MultiJava also supports open classes and value dispatching. Both parasitic methods and Half & Half support the use of inter-

faces as specializers in multimethods. Because it is difficult to modularly check multimethod ambiguity in the presence of interface specializers, parasitic methods modify the multimethod dispatching semantics so that ambiguities cannot exist, employing the textual order of methods to break ties. Half & Half resolves the problem by performing implementation-side typechecking on entire packages at a time, rather than on individual classes. For such package-level checking to be safe, Half & Half must also limit the visibility of some interfaces to their associated packages, thereby disallowing outside clients from employing those interfaces as specializers. In addition to multimethods, Half & Half supports a restricted form of *retroactive abstraction*, the ability to add new superclasses and superinterfaces to existing classes. Again, this is possible because of Half & Half's package-granularity typechecking. It is unclear how to simultaneously support Java's modular static typechecking and retroactive abstraction, so we have not yet added this feature to MultiJava.

Others have incorporated multimethods into Java without extending the language's syntax, instead using a library solution. These solutions have the advantage that any Java compiler can be used to compile programs that employ multiple dispatch. However, these solutions are typically not as general as MultiJava's multimethods. In addition, because multimethods are not typechecked specially, incompletenesses and ambiguities are not statically detected. For example, Forax et al. [2000] provide a `MultiMethod` class, and multimethods are declared by invoking `MultiMethod.create`. Grothoff [Grothoff 2003] provides an abstract `Runabout` class; user-defined subclasses of `Runabout` have a `visit` method that employs multiple dispatch. Both of these strategies rely heavily on Java's reflection in order to implement multimethod dispatch. By building dispatchers dynamically, based on the current set of loaded multimethods, these solutions can generate more efficient code than that generated by MultiJava's modular compilation strategy.

Dutchyn et al. [2001] also employ a library solution for incorporating multimethods into Java. They use a marker interface to indicate classes where static overloading should instead be treated as dynamic overriding (i.e., multimethod specialization). A modified virtual machine implements the changed semantics for classes bearing the marker interface. They show that the approach results in a large speed-up over double dispatching, because multimethod dispatch is done in native code. It would be interesting to consider combining the modified virtual machine approach with MultiJava. MultiJava's generated dispatcher code would be used to achieve the correct semantics when a MultiJava class was run on a standard virtual machine. A MultiJava-aware virtual machine could use native dispatch code, à la Dutchyn et al., to achieve faster execution, perhaps by using our existing multimethod bytecode attributes described in Section 4.4.3.

The Nice programming language [Bonniot and Keller 2003] is a recent object-oriented programming language that is similar to Java but has its heritage in $ML_{\leq}$ [Bourdoncle and Merz 1997]. Nice was developed after MultiJava and it includes multiple dispatch and open classes. Nice also includes a restricted form of retroactive abstraction based on abstract interfaces [Bonniot 2003]. Nice does not support modular implementation-side typechecking, which is the key technical contribution of our work. While MultiJava is designed to be a backward-compatible extension to

Java, Nice is a separate language with significant differences and incompatibilities. Nice is, however, designed to interoperate with Java programs and libraries, and the Nice compiler targets the standard Java virtual machine.

Ernst *et al.* [1998] describe a generalization of multimethod dispatch called *predicate dispatch*. Each method can be associated with a predicate guard, which specifies when the method is applicable. An object-oriented-style dispatching semantics is used, with logical implication of predicates as the specificity relation among methods. The authors provide a conservative implementation-side typechecking algorithm for predicate dispatch. The algorithm is non-modular, requiring access to the entire program to ensure safety. Follow-on work described efficient implementation techniques for predicate dispatch [Chambers and Chen 1999]. These techniques could also be useful for improving MultiJava's compilation strategy. JPred [Millstein 2004] is an extension of Java supporting predicate dispatch. JPred adapts and generalizes MultiJava's typechecking requirements to support modular, static typechecking for predicate dispatch.

8.2 Other Solutions to the Extensibility Problem

Jiazzi [McDirmid et al. 2001] is an extension to Java that adds a module mechanism based on *units* [Flatt and Felleisen 1998; Findler and Flatt 1999], a powerful form of parameterized module. Jiazzi supports extensibility idioms not provided by MultiJava, such as the ability to implement a *mixin* [Bracha and Cook 1990; Findler and Flatt 1999; Flatt et al. 1998], which is a class parameterized by its superclass. The authors also show how to encode an *open class pattern* in Jiazzi, whereby a module imports a class and exports a version of that class modified to contain a new method or field. Open classes in MultiJava allow two clients of a class to augment the class in independent ways, without having to be aware of one another. In contrast, in Jiazzi there must be a single module that integrates all augmentations, thereby creating the final version of the class. Module linking in Jiazzi is performed statically, so it is not possible to dynamically add new methods to existing classes. Dynamic augmentation is possible in MultiJava, since open classes are integrated with Java's regular dynamic loading process.

Zenger and Odersky [2001] describe an extensible datatype mechanism in the context of an object-oriented language. Classes can declare "cases," which are similar to ML data variants. Methods of other classes use functional-style pattern matching to dispatch on the data variants of an existing datatype. The result is a form of augmenting method similar to our noninvasive visitors. To ensure completeness in the presence of datatype extension, all methods that pattern-match on extensible datatypes must include the equivalent of the unspecialized method required by MultiJava's requirement R1. As with noninvasive visitors, Zenger and Odersky's functions are not extensible. Therefore, if new data variants require overriding function cases, a new function must be created that inherits the existing function cases, and clients must be modified to invoke the new function.

More recently, Zenger and Odersky [2005] present two solutions to the augmenting method problem written in the Scala programming language. Their solutions use three unique features of Scala: dependent types, mixin composition, and explicitly typed self references. Their solutions are dual to each other, with one primarily expressed in the object-oriented style and the other in a procedural style. Un-

like the MultiJava solution, where default operations must be provided to prevent unseen incompleteness, the Scala type system rules out invocation of operations on datatypes for which the operations are not defined. The trade-offs include a more complex type system in Scala and advance planning for extension of the base datatype or operations. Unlike MultiJava, which is a simple extension to Java, Scala is a unique programming language. Thus, it solves a different problem and is not subject to the same constraints as MultiJava. Scala's type system is based on the νObj calculus [Odersky et al. 2003]. Zenger and Odersky also demonstrate a solution to the binary method problem using Scala. Their solution is based on the double-dispatch technique and requires the same tedious coding. However, unlike traditional double dispatch, the use of dependent types allows (with advance planning) modular extension of the associated method family. Scala is designed to interoperate with both the Java Virtual Machine and libraries, and with the .NET libraries and runtime [Troelsen 2003].

Other work on the extensibility problem has addressed it in the context of functional languages. Extensible ML (EML) [Millstein et al. 2002] is an ML-like language that supports hierarchical, extensible datatypes and functions. Such constructs allow for the easy addition of both new data variants and new operations to existing abstractions. EML retains fully modular typechecking by adapting MultiJava's typechecking requirements. Garrigue shows how to use *polymorphic variants*, which are variants defined independently of any particular datatype, to obtain both modular data-variant and function extensibility in ML [Garrigue 2000]. However, unlike in MultiJava, both kinds of extensibliity require advance planning. Mixin modules [Duggan and Sourelis 1996] allow datatype and function declarations to be split across multiple modules, thereby providing a form of extensible datatypes and functions. Mixin modules must be explicitly combined to form the complete datatypes and functions. Therefore, there must be a single place in the program where all extensions to a given datatype or function are known. This contrasts with the "nonlinear" extensibility of MultiJava: there need not be a single compilation unit where all of a class's subclasses or all of a method family's methods are visible.

8.3 Advanced Separation of Concerns

Separation of concerns is the well-known software engineering concept that code for different subdomains, or aspects, of a problem should be made as independent as possible to encourage comprehensibility and efficiency (in both reuse and parallel development) [Parnas 1972; 1975]. Object-oriented languages encourage the separation of concerns into code representing individual classes in a model of the problem domain. However, there are some aspects which cut across the decomposition of a problem domain into classes [Harrison and Ossher 1993; Kiczales et al. 1997; Tarr et al. 1999]. The subfield dealing with this problem is known as advanced separation of concerns, or aspect-oriented software development.

Recently several languages have emerged that provide direct support for advanced separation of concerns. For example, AspectJ [Kiczales et al. 2001; AspectJ Team 2004] is an aspect-oriented extension to Java, whose *aspects* can extend existing classes in powerful ways. Hyper/J [Ossher and Tarr 2001] is a subject-oriented [Harrison and Ossher 1993] extension to Java that provides *hyperslices*, which are partially implemented modules that are composed to form classes or other mod-

ules. Both languages support open classes; for example, this ability corresponds to AspectJ's *inter-type declarations*. The languages additionally support many more flexible extensibility mechanisms than MultiJava. For example, AspectJ's *before* and *after* advice provide ways of modifying existing methods externally. To cope with this level of expressiveness, these languages employ non-modular typechecking and compilation strategies. For example, AspectJ's compiler "weaves" the aspects into their associated classes; only when all aspects that can possibly affect a class are available for weaving are final typechecking and compilation performed. (In recent versions of AspectJ, weaving can be performed on bytecode and typechecking is divided into separate phases, one performed incrementally at compile time and another whole-program phase performed during bytecode weaving.) Also, because of the weaving process, all clients of a woven class see the changes introduced by the aspects, unlike MultiJava's client-specific open classes.

Binary Component Adaptation (BCA) [Keller and Hölzle 1998] allows programmers to define *adaptation specifications*, which are directives for modifying existing classes. Such specifications can include the addition of new methods to existing classes, thereby supporting a form of open classes. Adaptation specifications can also include modifications not supported by MultiJava, like retroactive abstraction. The typechecking and compilation strategy of BCA is similar to the aspect-weaving approach described above, requiring access to all adaptation specifications that can affect a given class in order to typecheck and compile the class. The authors describe an on-line implementation of BCA, whereby the weaving is performed dynamically using a specialized class loader.

9. CONCLUSION

In this paper we have described the MultiJava programming language, motivated the design of the language, and discussed its modular, static typechecking and modular compilation strategy. MultiJava adds the ability to dispatch on a class externally, that is without modifying the class in place. Among other things, this allows MultiJava to cleanly solve the binary and augmenting method problems. It does so, unlike other solutions, without requiring advance planning by the original implementor of a datatype. We think that MultiJava represents a sweet spot in the design space: it allows the concise, declarative expression of multimethod dispatch and augmenting methods; and its type system is simple and intuitive.

We have also demonstrated how MultiJava's conservative extension of Java has allowed our users to easily adopt the language while creating more readable and maintainable code. Sample code drawn from our user community illustrates the benefits afforded by multiple dispatch and open classes and provides at least anecdotal evidence arguing for the inclusion of these features in mainstream programming languages.

ACKNOWLEDGMENTS

Thanks to David Cok for his work on the `mjdoc` documentation tool. Thanks to the members of the Spring 2003 Computer Science Writer's Workshop at Iowa State University—especially to Becca Wemhoff—and the anonymous referees for their helpful comments. Finally, many thanks to our users for their enthusiastic

adoption of MultiJava, insights on the language and possible improvements, and creative use beyond what we had envisioned.

REFERENCES

Agrawal, R., DeMichiel, L. G., and Lindsay, B. G. 1991. Static type checking of multi-methods. In *OOPSLA '91 Conference Proceedings*, A. Paepcke, Ed. ACM SIGPLAN Notices, vol. 26(11). ACM, New York, NY, 113–128.

Ancona, D., Lagorio, G., and Zucca, E. 2002. A formal framework for Java separate compilation. In *Proceedings of the 2002 European Conference on Object-Oriented Programming*. LNCS 2374. Springer-Verlag, Malaga, Spain.

Arnold, K., Gosling, J., and Holmes, D. 2000. *The Java Programming Language Third Edition*, Third ed. Addison-Wesley, Reading, MA.

Arnstein, L., Hung, C.-Y., Franza, R., Zhou, Q. H., Borriello, G., Consolvo, S., and Su, J. 2002. Labscape: A smart environment for the cell biology laboratory. *IEEE Pervasive Computing 1,* 3 (July), 13–21.

AspectJ Team. 2004. The AspectJ programming guide. Available from `http://eclipse.org/aspectj`.

Baumgartner, G., Jansche, M., and Läufer, K. 2002. Half & Half: Multiple dispatch and retroactive abstraction for Java. Tech. Rep. OSU-CISRC-5/01-TR08, Department of Computer Science, The Ohio State University. Mar.

Bonniot, D. 2003. Using kinds to type partially-polymorphic methods. In *Electronic Notes in Theoretical Computer Science*, G. Barthe and P. Thiemann, Eds. Vol. 75. Elsevier, New York, NY.

Bonniot, D. and Keller, B. 2003. The Nice user's manual. `http://nice.sourceforge.net`.

Bourdoncle, F. and Merz, S. 1997. Type-checking higher-order polymorphic multi-methods. In *Conference Record of POPL '97: the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, 302–315.

Boyland, J. and Castagna, G. 1997. Parasitic methods: Implementation of multi-methods for Java. In *Conference Proceedings of OOPSLA '97*. ACM SIGPLAN Notices, vol. 32(10). ACM, New York, NY, 66–76.

Bracha, G. and Cook, W. 1990. Mixin-based inheritance. In *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz, Ed. ACM SIGPLAN Notices, vol. 25(10). ACM, New York, NY, 303–311.

Bridges, S., Figueroa, M., Hsu, D., and Diorio, C. 2003. Field-programmable learning arrays. In *Advances in Neural Information Processing Systems 15*. MIT Press, Cambridge, MA.

Bruce, K., Cardelli, L., Castagna, G., Group, T. H. O., Leavens, G. T., and Pierce, B. 1995. On binary methods. *Theory and Practice of Object Systems 1,* 3, 221–242.

Cardelli, L. 1988. A semantics of multiple inheritance. *Information and Computation 76,* 2/3 (February/March), 138–164. A revised version of the paper that appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pages 51–66.

Castagna, G. 1994. Covariance and contravariance: conflict without a cause. Tech. Rep. liens-94-18, LIENS. Oct. Available by anonymous ftp from `ftp.ens.fr` in file `/pub/dmi/users/castagna/covariance.dvi.Z`. To appear in ACM TOPLAS, volume 17, number 3, March 1995.

Castagna, G. 1995. Covariance and contravariance: conflict without a cause. *ACM Trans. Program. Lang. Syst. 17,* 3, 431–447.

Castagna, G. 1997. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkhauser, Boston.

Castagna, G., Ghelli, G., and Longo, G. 1995. A calculus for overloaded functions with subtyping. *Information and Computation 117,* 1 (Feb.), 115–135. A preliminary version appeared in *ACM Conference on LISP and Functional Programming*, June 1992 (pp. 182–192).

Chambers, C. 1992. Object-oriented multi-methods in Cecil. In *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, O. L. Madsen, Ed. Lecture Notes in Computer Science, vol. 615. Springer-Verlag, New York, NY, 33–56.

Chambers, C. 1997. The Cecil language specification and rationale: Version 2.1. Available from `http://www.cs.washington.edu/research/projects/cecil/www/pubs/cecil-spec.html`.

Chambers, C. 1998. Towards Diesel, a next-generation OO language after Cecil. Invited talk, the *Fifth Workshop of Foundations of Object-Oriented Languages*, San Diego, California.

Chambers, C. and Chen, W. 1999. Efficient multiple and predicate dispatching. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA '99)*. ACM SIGPLAN Notices, vol. 34(10). ACM, New York, NY, 238–255.

Chambers, C. and Leavens, G. T. 1995. Typechecking and modules for multi-methods. *TOPLAS 17,* 6 (Nov.), 805–843.

Clifton, C. 2001. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Tech. Rep. 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, 50011. Nov. Available from `www.multijava.org`.

Clifton, C., Leavens, G. T., Chambers, C., and Millstein, T. 2000. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM SIGPLAN Notices, vol. 35(10). ACM, New York, NY, 130–145.

Cook, W. R. 1991. Object-oriented programming versus abstract data types. In *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, J. W. de Bakker, W. P. de Roever, and G. Rozenberg, Eds. Lecture Notes in Computer Science, vol. 489. Springer-Verlag, New York, NY, 151–178.

Drossopoulou, S., Eisenbach, S., and Wragg, D. 1999. A fragment calculus — towards a model of separate compilation, linking and binary compatibility. In *Logic in Computer Science*. IEEE, Trento, Italy, 147–156.

Duggan, D. and Sourelis, C. 1996. Mixin modules. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*. ACM SIGPLAN Notices, vol. 31(6). ACM, ACM Press, New York, NY, 262–273.

Dutchyn, C., Szafron, D., Bromling, S., and Holst, W. 2001. Multi-dispatch in the java virtual machine: Design and implementation. In *Sixth Conference on Object-Oriented Technologies and Systems (COOTS)*. USENIX, Berkeley, CA.

Ernst, M. D., Kaplan, C., and Chambers, C. 1998. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98: 12th European Conference on Object-Oriented Programming, Brussels, Belgium*. Lecture Notes in Computer Science, vol. 1445. Springer-Verlag, New York, NY, 186–211.

Feinberg, N., Keene, S. E., Mathews, R. O., and Withington., P. T. 1997. *The Dylan Programming Book*. Addison-Wesley Longman, Reading, Mass.

Findler, R. B. and Flatt, M. 1999. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. ACM SIGPLAN Notices, vol. 34(1). ACM, New York, NY, 94–104.

Flatt, M. and Felleisen, M. 1998. Units: Cool modules for hot languages. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM SIGPLAN Notices, vol. 33(5). ACM, New York, NY, 236–248.

Flatt, M., Krishnamurthi, S., and Felleisen, M. 1998. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, 171–183.

Forax, R., Duris, E., and Roussel, G. 2000. Java multi-method framework. In *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS '00), Sydney, Australia*. IEEE Computer Society Press, Los Alamitos, California.

Gagnon, E. and Hendren, L. J. 1998. SableCC, an object-oriented compiler framework. In *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS '98), Santa Barbara, California*. IEEE Computer Society Press, Los Alamitos, California.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass.

Garrigue, J. 2000. Code reuse through polymorphic variants. In *Proceedings of the Workshop on Foundations of Software Engineering*. Sassaguri, Japan. Available from `http://wwwfun.kurims.kyoto-u.ac.jp/~garrigue/papers/fose2000.html`.

Goldberg, A. 1984. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley Publishing Co., Reading, Mass.

Gosling, J., Joy, B., Steele, G., and Bracha, G. 2000. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass.

Grimm, R., Davis, J., Lemar, E., MacBeth, A., Swanson, S., Gribble, S., Anderson, T., Bershad, B., Borriello, G., and Wetherall, D. June 2001. Programming for pervasive computing environments. Tech. Rep. UW-CSE-01-06-01, Department of Computer Science and Engineering, University of Washington.

Grothoff, C. 2003. Walkabout revisited: The Runabout. In *Proceedings of the 2003 European Conference on Object-Oriented Programming*. LNCS 2743. Springer-Verlag, Darmstadt, Germany.

Harrison, W. and Ossher, H. 1993. Subject-oriented programming (a critique of pure objects). In *OOPSLA 1993 Conference Proceedings*, A. Paepcke, Ed. ACM SIGPLAN Notices, vol. 28(10). ACM, New York, NY, 411–428.

Hightower, J., Brumitt, B., and Borriello, G. 2002. The location stack: A layered model for location in ubiquitous computing. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems & Applications (WMCSA 2002)*. IEEE Computer Society Press, Callicoon, NY, 22–28.

Ingalls, D. H. H. 1986. A simple technique for handling multiple polymorphism. In *OOPSLA '86 Conference Proceedings*, N. Meyrowitz, Ed. ACM SIGPLAN Notices, vol. 21(11). ACM, New York, NY, 347–349.

Keller, R. and Hölzle, U. 1998. Binary component adapatation. In *ECOOP '98–Object-Oriented Programming*, E. Jul, Ed. LNCS. Springer, New York, NY, 307–329.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. 2001. An overview of AspectJ. In *ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary*, J. L. Knudsen, Ed. Lecture Notes in Computer Science, vol. 2072. Springer-Verlag, Berlin, 327–353.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. 1997. Aspect-oriented programming. In *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, M. Akşit and S. Matsuoka, Eds. Lecture Notes in Computer Science, vol. 1241. Springer-Verlag, New York, NY, 220–242.

Kopi 2004. Kopi project home page. `http://www.dms.at/kopi`.

Krishnamurthi, S., Felleisen, M., and Friedman, D. P. 1998. Synthesizing ojbect-oriented and functional design to promote re-use. In *ECOOP'98—Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, E. Jul, Ed. Lecture Notes in Computer Science, vol. 1445. Springer-Verlag, New York, NY, 91–113.

Leavens, G. T. and Millstein, T. D. 1998. Multiple dispatch as dispatch on tuples. In *OOPSLA '98 Conference Proceedings*. ACM SIGPLAN Notices, vol. 33(10). ACM, New York, NY, 374–387.

Lee, K., LaMarca, A., and Chambers, C. 2003. HydroJ: Object-oriented pattern matching for evolvable distributed systems. In *Proceedings of the 2003 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM SIGPLAN Notices, vol. 38(11). ACM, New York, NY, 205–223.

Liang, S. and Bracha, G. 1998. Dynamic class loading in the Java virtual machine. In *OOPSLA '98 Conference Proceedings*. ACM SIGPLAN Notices, vol. 33(10). ACM, ACM, New York, NY, 36–44.

Lindholm, T. and Yellin, F. 2000. *The Java Virtual Machine Specification*, Second ed. Addison-Wesley Publishing Co., Reading, MA.

Litvinov, V. 1998. Constraint-based polymorphism in Cecil: Towards a practical and static type system. In *OOPSLA '98 Conference Proceedings*. ACM SIGPLAN Notices, vol. 33(10). ACM,

New York, NY, 388–411. The proceedings mistakenly contain a preliminary version of the paper. The final version is at `ftp://ftp.cs.washington.edu/pub/chambers/sbp-oopsla.ps.gz`.

Martin, P. 1998. Java, the good, the bad and the ugly. *ACM SIGPLAN Notices 33,* 4 (Apr.), 34–39.

McDirmid, S., Flatt, M., and Hsieh, W. 2001. Jiazzi: New-age components for old-fashioned Java. In *Proceedings of OOPSLA '01 Conference on Object-Oriented Programming, Languages, Systems, and Applications*. SIGPLAN Notices, vol. 36(11). ACM, New York, NY, 211–222.

Millstein, T. 2003. Reconciling software extensibility with modular program reasoning. Ph.D. thesis, Department of Computer Science & Engineering, University of Washington.

Millstein, T. 2004. Practical predicate dispatch. In *Proceedings of the OOPSLA '04 conference on Object Oriented Programming Systems Languages and Applications*. ACM SIGPLAN Notices, vol. 39(11). ACM, New York, NY, 345–364.

Millstein, T., Bleckner, C., and Chambers, C. 2002. Modular typechecking for hierarchically extensible datatypes and functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM SIGPLAN Notices, vol. 37(9). ACM, New York, NY, 110–122.

Millstein, T. and Chambers, C. 1999. Modular statically typed multimethods. In *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, R. Guerraoui, Ed. Lecture Notes in Computer Science, vol. 1628. Springer-Verlag, New York, NY, 279–303.

Millstein, T. and Chambers, C. 2002. Modular statically typed multimethods. *Information and Computation 175,* 1 (May), 76–118.

Millstein, T., Reay, M., and Chambers, C. 2003. Relaxed MultiJava: Balancing extensibility and modular typechecking. In *Proceedings of the 2003 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM SIGPLAN Notices, vol. 38(11). ACM, New York, NY, 224–240.

Mugridge, W. B., Hosking, J. G., and Hamer, J. 1991. Multi-methods in a statically-typed programming language. In *ECOOP '91 Conference Proceedings, Geneva, Switzerland*, P. America, Ed. Lecture Notes in Computer Science, vol. 512. Springer-Verlag, New York, NY.

Nordberg, M. E. 1998. Default and extrinsic visitor. In *Pattern Languages of Program Design 3*, R. C. Martin, D. Riehle, and F. Buschmann, Eds. Addison-Wesley Publishing Co., Reading, MA, 105–123.

Nystrom, N., Clarkson, M. R., and Myers, A. C. 2003. Polyglot: An extensible compiler framework for java. In *Proceedings of CC 2003: 12'th International Conference on Compiler Construction*. Springer-Verlag, New York, NY.

Odersky, M., Cremet, V., Röckl, C., and Zenger, M. 2003. A nominal theory of objects with dependent types. In *ECOOP 2003, European Conference on Object-Oriented Programming, Darmstadt, Germany*. Springer-Verlag, New York, NY, 201–224.

Odersky, M. and Wadler, P. 1997. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, 146–159.

Ossher, H. and Tarr, P. 2001. Using multidimensional separation of concerns to (re)shape evolving software. *Commun. ACM 44,* 10 (Oct.), 43–50.

Paepcke, A. 1993. *Object-Oriented Programming: The CLOS Perspective*. The MIT Press, Boston, Mass.

Palsberg, J. and Jay, C. B. 1998. The essence of the visitor pattern. In *Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC*. IEEE, Vienna, Austria, 9–15.

Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15,* 12 (Dec.), 1053–1058.

Parnas, D. L. 1975. Software engineering or methods for the multi-person construction of multi-version programs. In *Programming Methodology, 4th Informatik Symposium, IBM Germany, Wildbad, 25-27 September, 1974*, C. E. Hackl, Ed. Lecture Notes in Computer Science, vol. 23. Springer-Verlag, New York, NY, 225–235.

Philipose, M., Fishkin, K. P., Perkowitz, M., Patterson, D. J., Fox, D., Kautz, H., and Hahnel, D. 2004. Inferring Activities from Interactions with Objects. *Pervasive Computing Magazine 3,* 4, 10–17.

Rain 2004. Rain home page. `http://seattleweb.intel-research.net/projects/rain`.

Reynolds, J. C. 1975. User-defined types and procedural data structures as complementary approaches to type abstraction. In *New Directions in Algorithmic Languages*, S. A. Schuman, Ed. IRIA, Rocquencourt, 157–168.

Reynolds, J. C. 1980. Using category theory to design implicit conversions and generic operators. In *Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark*, N. D. Jones, Ed. Lecture Notes in Computer Science, vol. 94. Springer-Verlag, New York, NY, 211–258.

Shalit, A. 1997. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass.

Steele Jr., G. L. 1990. *Common LISP: The Language*, Second ed. Digital Press, Bedford, Mass.

Stroustrup, B. 1997. *The C++ Programming Language: Third Edition*. Addison-Wesley Publishing Co., Reading, Mass.

Tarr, P. L., Ossher, H., Harrison, W. H., and Sutton Jr., S. M. 1999. *N* degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*. ACM, New York, NY, 107–119.

Torgersen, M. 2004. The expression problem revisited: Four new solutions using generics. In *ECOOP '04 - Object-Oriented Programming European Conference*, M. Odersky, Ed. Lecture Notes in Computer Science, vol. 3086. Springer-Verlag, New York, NY, 123–143.

Troelsen, A. 2003. *C# and the .NET platform*, 2nd ed. Apress, Berkeley, CA.

Vlissides, J. 1999. Visitor in frameworks. *C++ Report 11,* 10 (November/December), 40–46.

Zenger, M. and Odersky, M. 2001. Extensible algebraic datatypes with defaults. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*. ACM SIGPLAN Notices, vol. 36(10). ACM, ACM, New York, NY, 241–252.

Zenger, M. and Odersky, M. 2005. Independently extensible solutions to the expression problem. In *The 12th International Workshop on Foundations of Object-Oriented Languages (FOOL 12)*. ACM, Long Beach, California.

Received January 2004; December 2004